Universidad Autónoma de Madrid

Escuela Politécnica Superior

Grado en Ingeniería Informatica

TRABAJO DE FIN DE GRADO

ANÁLISIS DE LA ARQUITECTURA DIRIGIDA POR EVENTOS (EDA)

Joaquín Verdejo Salvador
Tutor: Sergio Gómez López
Ponente: Simone Santini

abril 2020

ANÁLISIS DE LA ARQUITECTURA DIRIGIDA POR EVENTOS (EDA)

Autor: Joaquín Verdejo Salvador
Tutor: Sergio Gómez López
Ponente: Simone Santini

Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid

abril 2020

Agradecimientos

A mi familia, por todo lo que me han apoyado a lo largo de toda mi vida y ayudándome en las decisiones importantes.

A mis amigos del colegio que a pesar de las dificultades para vernos, por los diferentes, caminos tomados siempre están ahí cuando se les necesita.

A mis amigos de la carrera que con su ayuda y apoyo hemos podido sacar adelante todas las asignaturas y además disfrutar de la vida universitaría y así desconectar un poco.

A los profesores del Grado de Ingeniería Informática de la EPS, y el resto de trabajadores de la UAM por toda la ayuda ofrecida para llegar hasta aquí

A mis compañeros de Indra, por todo lo que he aprendido de ellos y toda la ayuda que me han dado, formando un gran equipo.

Resumen

Resumen — En la actualidad las empresas tecnológicas están tendiendo hacia la arquitectura basada en microservicios frente a arquitecturas monolíticas. Esta nueva arquitectura a pesar de ser algo más compleja de desarrollar, soluciona muchos de los problemas de los programas monolíticos, como la localización de errores, el despliegue, el trabajo con mucho desarrolladores, entre otros problemas. Aunque los microservicios nos plantean otros dilemas, como puede ser la comunicación entre todos los módulos que forman la aplicación.

A lo largo de este Trabajo de Fin de Grado se va a realizar un análisis de la Arquitectura Basada en Eventos, una forma de comunicación entre microservicios alternativa a la comunicación via REST, forma de comunicación predominante actualmente. Se puede dividir en dos partes, por un lado habrá un análisis más teórico y por otro lado se implementarán dos ejemplos de uso y de esta manera poder hacer un mejor análisis y ver si merece la pena la arquitectura en situaciones reales. El proyecto se ha realizado para la empresa Indra Sistemas, S.A, más específicamente para el departamento de arquitectura. Que incluirá parte de la documentación del análisis en su Marco de Referencia y se incluirán los ejemplos implementados como aceleradores para futuras implantaciones de la arquitectura.

Esta arquitectura se basa en la generación de eventos al producirse un cambio de estado en el sistema, estos eventos se envían a un gestor de eventos que se encargará de transmitir los mensajes al resto de microservicios que estén suscritos y que se encargarán de procesarlos y realizar las operaciones pertinentes.

En la primera parte se ha realizado un estudio de esta arquitectura buscando información teórica sobre la implantación y a través de esta información sacar conclusiones, ventajas, desventajas de su uso. Se harán diversas comparaciones con la que es su alternativa más directa, las peticiones REST y de esta manera se verá en que momentos es mejor usar un método u otro basándonos en complejidad, resistencia a fallos, consumos de recursos, etc. Además veremos las tecnologías con las que se puede implantar y la recomendaciones personales.

En la segunda parte se han desarrollado dos módulos. Por un lado tenemos un sistema de ejemplo, que hará uso de EDA para la comunicación y kafka como gestor de eventos. Está formado por un productor que al tratarse de un ejemplo será un simulador que generará eventos con información aleatoria con una estructura basada en estado de vuelos. A parte tenemos varios consumidores que procesarán estos eventos. Todos ellos desarrollados en Spring. Por otro lado tenemos un servicio que actuará como un monitor del gestor de eventos, en el que podremos ver estado, mensajes, temas y consumidores. De esta forma podremos ver diferentes métricas de una manera visual, sin tener que entrar en la configuración de nuestro gestor de eventos y obtener información del sistema.

Palabras clave — arquitectura, microservicios, eventos, comunicación, kafka, Spring, EDA

Abstract

Abstract — Currently, technology companies are tending towards architecture based on microservices as opposed to monolithic architectures. This new architecture, despite being somewhat more complex to develop, solves many of the problems of monolithic programs, such as error localization, deployment, working with many developers, among other problems. Although microservices pose other dilemmas, such as communication between all the modules that make up the application.

Throughout this Final Degree Project, we are going to analyse the Event Based Architecture, an alternative form of communication between microservices to communication via REST, currently the predominant form of communication. It can be divided into two parts, on the one hand there will be a more theoretical analysis and on the other hand two usage examples will be implemented and in this way we can do a better analysis and see if the architecture is worthwhile in real situations. The project has been carried out for the company Indra Sistemas, S.A, more specifically for the architecture department. That will include part of the analysis documentation in its Reference Framework and the examples implemented as accelerators for future architecture implementations will be included.

This architecture is based on the generation of events when a state change occurs in the system, these events are sent to an event manager that will be in charge of transmitting the messages to the rest of the subscribed microservices that will be in charge of process them and carry out the pertinent operations.

In the first part, a study of this architecture has been carried out looking for theoretical information on the implantation and through this information to draw conclusions, advantages, disadvantages of its use. Various comparisons will be made with what is your most direct alternative, REST requests and in this way you will see when it is better to use one method or another based on complexity, resistance to failures, consumption of resources, etc. We will also see the technologies with which it can be implanted and personal recommendations.

In the second part, two modules have been developed. On the one hand we have an example system, which will use EDA for communication and kafka as the event broker. It is made up of a producer who, being an example, will be a simulator that will generate events with random information with a structure based on flight status. Besides we have several consumers who will process these events. All of them developed in Spring. On the other hand we have a service that will act as a monitor of the event manager, in which we can see status, messages, topics and consumers. In this way we can see different metrics in a visual way, without having to enter the configuration of our event manager and obtain system information.

Key words — architecture, microservices, events, communication, kafka, Spring, EDA

Índice general

Índice de figuras

1 Introducción

En este primer capítulo del TFG se va a mostrar una idea global del proyecto, mostrando cuales han sido las motivaciones para su realización, los objetivos marcados y las distintas fases por las que ha pasado el desarrollo.

1.1 Motivación

En los últimos años se ha visto una tendencia del mundo empresarial en basar los nuevos sistemas en microservicios frente a las arquitecturas monolíticas. Esta nueva arquitectura a pesar de ser más compleja de desarrollar trae consigo una gran cantidad de ventajas, principalmente en temas de escalabilidad, reusabilidad o despliegue entre otras. Pero viene acompañadas con otro tipo de decisiones que se deben tomar, una de ellas es el método de comunicación entre estos microservicios.

Esta comunicación es un punto clave en el correcto funcionamiento del sistema, ya que un fallo de comunicación entre microservicios provocará un fallo en las salidas esperadas. Por esta razón es importante emplear algo de tiempo en un análisis de los diferentes sistemas de comunicación que existen y de esta manera utilizar el más correcto en cada circunstancia.

En la actualidad, la gran mayoría de arquitecturas basadas en microservicios utilizan la comunicación via REST, que utiliza llamadas HTTP en las que están contenidas toda la información que se desea transmitir. Este protocolo tiene sus orígenes en el año 2000 y con las herramientas actuales es sencillo de implementar en los servicios y de una manera eficiente, por lo que parece lógico su implantación como protocolo de comunicación. Pero existen circunstancias en las que este protocolo nos presenta problemas debido a que es síncrono, es decir para la comunicación entre dos puntos ambos deben de estar disponibles, y existen ocasiones que esto no es así, ya sea por una elevada carga o por la caída de servidores. Es aquí donde entra la arquitectura dirigida por eventos(EDA), que busca solucionar los problemas de los sistema síncronos.

Por lo expuesto anteriormente desde el Departamento de Arquitectura de Minsait(Indra) se buscaba un análisis de EDA para poder conocer en que arquitecturas se recomienda su uso y así poder ofrecer arquitecturas más robustas y eficientes a los clientes del departamento. Ademas de recomendar algunas configuraciones para su implantación, es decir generar algo así como una estandarización y un punto de referencia.

1.2 Objetivos

Este trabajo de fin de grado busca realizar un estudio lo más exhaustivo posible de la arquitectura basada en eventos, y a partir de toda la información obtenida poder ofrecer un análisis lo más completo posible sobre dicha arquitectura. De este modo, poder obtener las principales ventajas y desventajas de su implantación y en qué circunstancias recomendar su uso.

A través de este análisis se pretenden mostrar algunas tecnologías que pueden ayudarnos a la implantación de esta arquitectura, así como mostrar las mejores configuraciones de estas tecnologías o las mejores estructuras a utilizar.

Aparte del análisis también se implementará un ejemplo del principal caso de uso de esta arquitectura, de esta manera se podrá ver de una forma un poco más visual y sobre todo de manera práctica el funcionamiento. Tanto el gestor de eventos como los ejemplos implementados se probarán tanto en local como en producción, para obtener información más real.

Para finalizar, una vez obtenido el análisis y el ejemplo, se implementará un microservicio que actuará como pantalla de monitorización del gestor de eventos, y así poder obtener estadísticas de los consumidores, productores y de los eventos.

1.3 Estructura del documento

El documentos se ha estructurado siguiendo las distintas etapas que se han seguido en el análisis:

- En el *Capítulo 1* se ha realizado una breve introducción del proyecto, la motivación para hacerlo y la estructura del documento
- En el *Capítulo 2* se va a hablar de manera general sobre los microservicios y la arquitectura basada en eventos.
- En el *Capítulo 3* se va a entrar más en detalle en las características de la arquitectura y diferencias frente a REST.
- En el *Capítulo 4* se presentan los principales casos de uso de la arquitectura.
- En el *Capítulo 5* se muestra el ejemplo implementado de la arquitectura.
- En el *Capítulo 6* se presenta el módulo de monitorización.
- En el *Capítulo 7* se realiza un resumen final del proyecto a modo de conclusiones.

2

¿Qué es la arquitectura Event-Driven?

Antes de comenzar con las características, ventajas y desventajas de la arquitectura basada en eventos, es importante tener una idea general de lo que son los microservicios y algunas de sus características, una vez que sabemos lo que son y su funcionamiento podremos entender mejor lo que es la arquitectura basada en eventos.

2.1 Microservicios

El modelo de microservicios [1] se basa en dividir la aplicación en componentes más pequeños que desempeñen una funcionalidad concreta, con la particularidad de que estos módulos serán programas independientes, con su propia configuración, seguridad, base de datos, etc. Es decir, en una aplicación monolítica como buena práctica de programación, se divide el programa en módulos o clases, que estarán en diferentes ficheros, pero a la hora de la compilación se nos generará un único ejecutable con toda la funcionalidad, esto es, a nivel interno estará dividido en módulos independientes, pero a ojos de los usuarios tendremos un único programa; en cambio en los microservicios cada uno de estos módulos será un programa independiente y el conjunto de funcionalidades de la aplicación final será fruto de un intercambio de información entre los distintos módulos que lo forman. Por lo tanto, en las aplicaciones tradicionales estábamos ante un único bloque que englobaba todas las funcionalidades y se compila y despliega de una vez, mientras que los microservicios son compilados y desplegados de manera independiente.

En el hecho de tener las funcionalidades divididas encontramos la principal ventaja de los microservicios y es que antes un cambio en una funcionalidad suponía el tener que compilar todo el conjunto, lo cual dependiendo del tamaño puede llegar a ser un proceso complejo y lo mismo ocurre con el despliegue, en cambio en los microservicios solo es necesario la compilación y despliegue del módulo que abarque la funcionalidad, esto se suele hacer mediante un correcto CI/CD por servicio [2]. Otra de las ventajas de estos módulos es la reutilización [3], ya que al abarcar estos módulos funcionalidades muy específicas y trabajar de manera independiente, se pueden utilizar en distintas aplicaciones con muy poco trabajo de adaptación. De esta

ultima ventaja podemos sacar otra más, que es una mejor protección frente a fallos y mayor velocidad de recuperación, ya que el fallo en un módulo(si está bien diseñado) solo afecta a una funcionalidad del programa y al tener que reparar y desplegar solo un módulo tendremos una rápida recuperación.

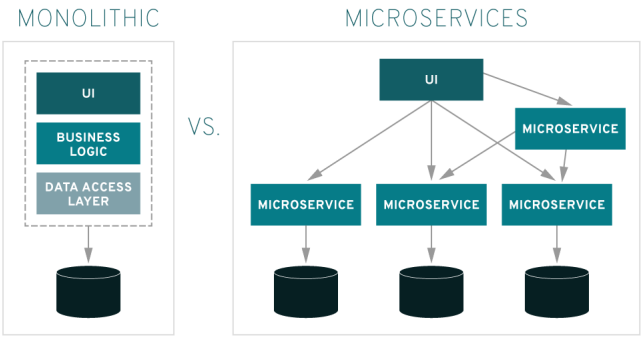

Figura 2.1: Microservicios vs Monolíticos
Fuente: https://www.redhat.com/es/topics/microservices/what-are-microservices

En una aplicación con una arquitectura monolítica, la comunicación interna consiste en llamar a diferentes clases, funciones, métodos, etc, pero siempre dentro de la propia aplicación, en cambio en los microservicios nos surge un problema y este es que las funcionalidades están fuera de la aplicación, es decir, la comunicación [4] es externa, por lo que deberemos salir del módulo o incluso del servidor para realizar el intercambio de información. Por lo que los desarrolladores se deben enfrentar a un nuevo problema que es realizar todas las conexiones entre todos los microservicios necesarios. Existen varias formas de realizar esta comunicación: HTTP/HTTPS, AMQP, TCP, etc... A día de hoy la más utilizada por su sencillez de implementación, su eficiencia y la seguridad que ofrece es el protocolo HTTP/HTTPS, que es un protocolo síncrono basado en el esquema de petición-respuesta, un servidor envía una petición a otro, y este responde con la información pedida o guarda la información enviada según lo deseado.

El protocolo HTTP/HTTPS hace que exista una dependencia entre los microservicios, ya que el emisor del mensaje debe esperar la respuesta del otro servicio, por lo que el servicio siempre debe estar disponible o de lo contrario nos encontraremos ante un error o un bloqueo del servicio. Y es aquí donde cobra importancia la arquitectura basada en eventos. Esta nueva forma de comunicarnos crea una total independencia entre módulos y a pesar de que un módulo este caído el sistema global podría seguir funcionando con normalidad, gracias a que convertimos la arquitectura de microservicios en una arquitectura asíncrona. En los siguientes puntos entraremos en detalle en que es y como funciona esta arquitectura, y los próximos capítulos entraremos en detalles en las características, ventajas, consejos de implementación y veremos ejemplos de esta arquitectura en funcionamiento y analizaremos estos ejemplos.

2.2 Arquitectura dirigida en eventos(EDA)

Ahora que ya tenemos una idea general de lo que son los microservicios y de como funcionan vamos a comenzar a hacer una introducción a la EDA [5] cómo método de comunicación entre ellos y a exponer las diferencias frente a arquitecturas tradicionalmente usadas.

2.2.1. Introducción

La arquitectura dirigida por eventos es un patrón de arquitectura que utiliza notificaciones de eventos como principal mecanismo de comunicación de información entre distintas aplicaciones o microservicios. Este patrón no es algo nuevo, lleva definido desde hace años, aunque nunca ha conseguido estar a la altura de estandarizaron de otros patrones de comunicación como pueden ser SOAP o REST, al menos desde el punto de vista de comunicación entre servicios, ya que el uso de eventos si lo hemos visto dentro de ciertas tecnologías como java Swing, o Android Studio, entre otras. En estos ejemplos, los eventos se usan de manera interna principalmente para poder reaccionar frente a acciones realizadas por el usuario.

Es importante destacar que esta arquitectura no viene a sustituir a otros patrones ya establecidos, sino a convivir junto con ellos, ya que no en todos los casos podrá valernos. Será tarea de los desarrolladores realizar un análisis previo.

Para tener una idea más clara de como funciona esta arquitectura y sus diferencias frente a un enfoque tradicional(REST) se han realizado dos esquemas de lo que sería una aplicación con tres microservicios, en el primero tenemos el enfoque tradicional mediante peticiones API REST y en el segundo mediante la comunicación por eventos. Además gracias al segundo esquema veremos las distintas partes esenciales que forman esta arquitectura.

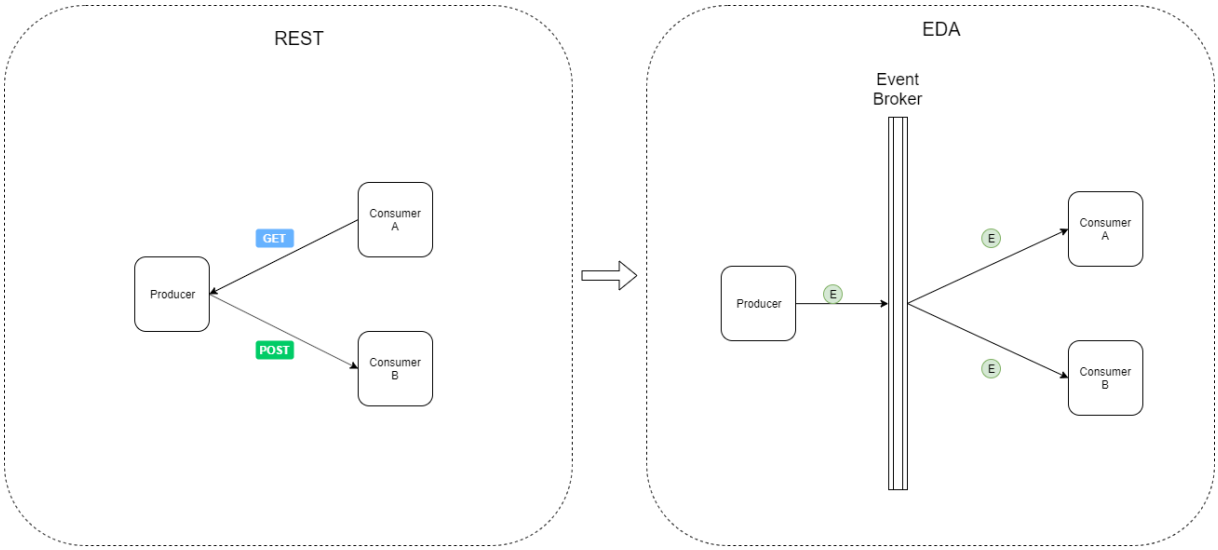

Figura 2.2: Arquitectura REST vs EDA

En la figura 2.2 podemos ver una misma arquitectura de una aplicación pero con dos enfoques distintos, y gracias a ella se ve claramente la principal diferencia. En el patrón REST vemos una comunicación directa entre los microservicios, con sus ventajas e inconvenientes que veremos más adelante; y en el patrón EDA vemos que la comunicación pasa por un intermediario, este es el Event Broker, que se encargará de gestionar y transmitir los eventos a sus respectivos destinos, por lo que no hay ningún tipo de comunicación directa entre nuestros servicios. En la imagen también podemos ver los distintas elementos en los que se basa la EDA: productor, evento, gestor de eventos y consumidor, los cuales explicaremos en la siguiente sección.

2.2.2. Elementos EDA

En esta sección vamos a explicar los 4 elementos imprescindibles en una arquitectura EDA. Podemos encontrar más elementos externos como pueden ser gestión de logs o dashboards para controlar, pero estos no serán no son imprescindibles para la arquitectura, tan solo servirán para obtener mejoras o un mayor control de la gestión de eventos.

Evento

Según la RAE un evento es: "Cosa que sucede" [6]. Esta definición puede parecer muy simple e incompleta, sin embargo es bastante válida ya que los productores de la arquitectura lanzarán eventos al detectar algún cambio en su estado. Aunque para matizarla un poco dentro de EDA, diremos que un evento es: un dato inmutable del estado de "algo" en un instante determinado y producido al ocurrir un cambio de estado. La característica más importante es "dato inmutable en un instante", con inmutable nos referimos a la modificación de los valores del evento por otros, ya que esta modificación crearía incoherencias, sin embargo, existe la posibilidad de añadir información durante un procesado. Estaríamos así ante un evento compuesto, ya que contendría, por un, lado la información del estado del sistema en un determinado momento y, además, le sumaríamos una información extra de un procesado posterior a la generación. Tras ello se volverá a lanzar al gestor para su consumo. Esto a pesar de ser posible no es recomendable, ya que puede provocar que tengamos información duplicada, debido a que en nuestro gestor tendremos ambos tipos de eventos, por un lado los generados previamente, y por otro los enriquecidos y ambos tendrán una parte de información idéntica, debido a la inmutabilidad de los mismos.

La estructura que se utilizará dependerá de cada desarrollador, aunque es importante que tenga ciertas características, como es un control de versiones y tener bien documentada la estructura utilizada. Aún así tras analizar varias estructuras utilizadas [7] en ejemplos y aplicaciones considero que la siguiente es la más correcta, además este es el formato que se va a recomendar internamente desde el departamento de arquitectura para la implementación de esta arquitectura en futuros proyectos:

```
{
  eventId : <ID>,
  eventName : '<EVENT_NAME>',
  creationDate : <CREATION_DATE>,
  author : '<AUTHOR>',
  version : <VERSION>,
  eventDataFormat  : <DATA_FORMAT>
  payload : '{
      "param1" : 'test',
      (...)
  }'

}
```

- *id*: Un identificador único que deberá tener un evento, nos ayudará durante los procesados, principalmente durante la gestión de errores para realizar búsquedas de forma rápida y sencilla.

- *eventName*: El nombre del evento, esto nos ayudará a agrupar un conjunto de eventos del misto tipo.

- *creationDate*: La marca de tiempo del momento de la creación, puede estar en el formato que se desee, pero es importante dejar claro este formato para que los productores no tengan problemas durante el procesado.

- *author*: Un identificador del sistema que genera el evento, puede ser el nombre del servicio, de un sensor o de un usuario en el caso que sean estos quien creen directamente los eventos.

- *version*: La versión de la estructura del evento, esto nos ayudará a que los consumidores puedan procesar eventos antiguos.

- *dataFormat*: El formato que tendrá la carga que contiene toda la información del evento. Más adelante veremos el recomendado.

- *payload*: La información del evento. Importante que esté en el formato indicado anteriormente y que la propia estructura interna de este campo esté documentada para que los consumidores no tengan problemas en su procesado.

Como se puede apreciar se ha elegido el formato JSON. Esto no tiene porque ser así, podría elegir cualquier otro formato de texto como podría ser XML, sin embargo tras un breve análisis de ambos se ha considerado que JSON es el formato idóneo para EDA, las principales ventajas que veo frente a EDA son:

- El formato JSON nos ofrece una mayor velocidad lectura, escritura y envío, esto se debe principalmente a su menor tamaño, haciendo pruebas con ficheros JSON y XML con el mismo contenido he calculado que XML es aproximadamente un 20 % más grande en cuanto a tamaño que JSON. Puesto que EDA deberá mover grandes cantidades de mensajes nos conviene que estos sean del menor tamaño posible. Esta es la principal característica que nos hace decantarme por JSON.

- JSON en un formato más legible para las personas frente a XML.

- Facilidad para añadir nuevos campos.

A parte de estas características que en mi opinión nos hace decantarnos por JSON, se puede ver que la tendencia global es el uso de JSON frente a XML. Esta tendencia la podemos comprobar a través de las búsquedas y consultas globales, a continuación muestro dos gráficas que lo muestran:

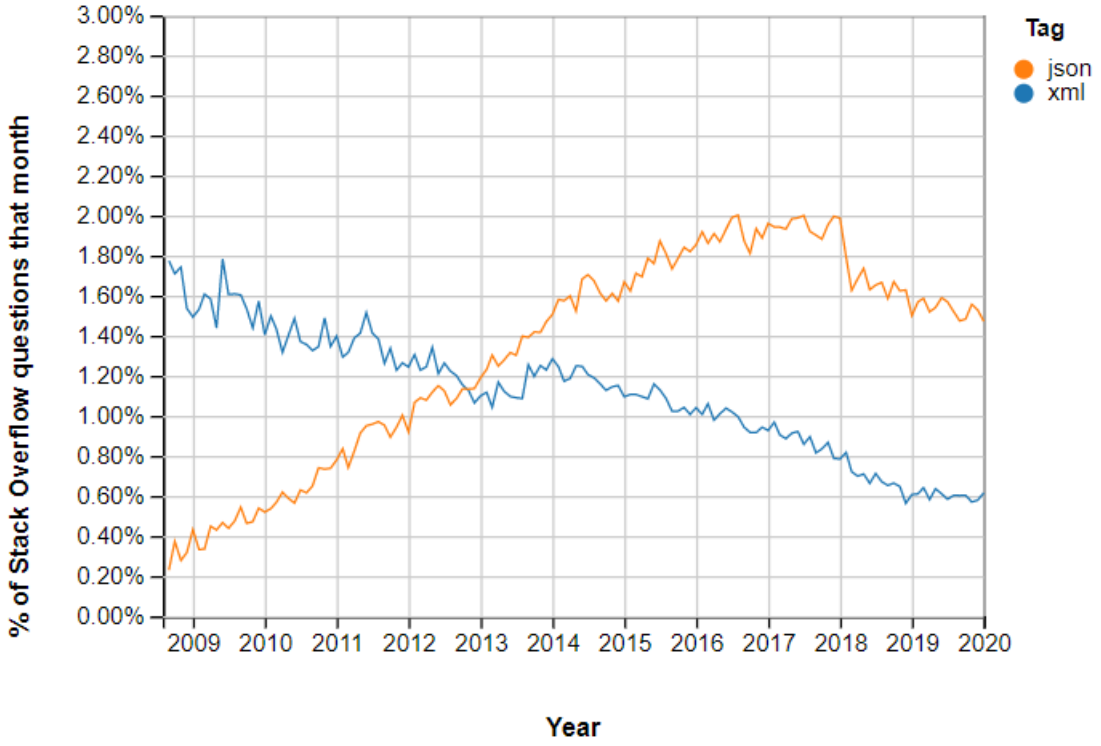

Figura 2.3: % de consultas en Stack Overflow JSON vs XML
Fuente: https://insights.stackoverflow.com/trends

En la figura 2.3 podemos observar el número de consultas de cada uno de los términos en la página web "Stack Overflow", desde 2009 hasta hoy. Esta página es una de las mayores comunidades donde desarrolladores informáticos realizan consultas sobre problemas de programación en diferentes lenguajes. Gracias a su página de tendencias, podemos ver como las preguntas sobre JSON van en aumento mientras que la consultas en XML se van reduciendo, con esta información se puede deducir que los desarrolladores está sustituyendo el XML frente al JSON.

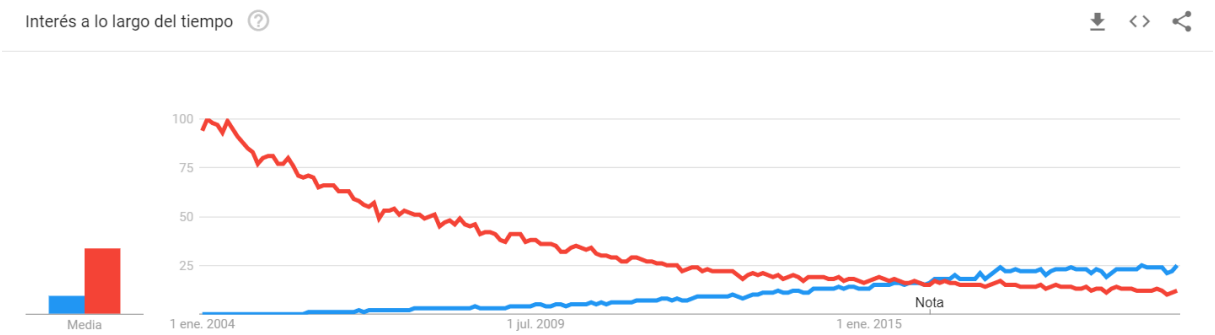

Figura 2.4: % de búsquedas en google JSON vs XML
Fuente: https://trends.google.es/trends/explore?q=%2Fm%2F05cntt,XML

Para corroborar y confirmar esta tendencia se ha realizado la misma consulta en la página de tendencias del buscador Google a nivel global, y en la 2.4 vemos como sigue exactamente la misma tendencia. Gracias a estas dos estadísticas que nos ofrecen ambas páginas podemos decir con total seguridad que los desarrolladores de manera general, prefieren usar JSON frente a XML.

A pesar de ello según la conveniencia del desarrollador para evitar incompatibilidades podría enviarse en XML o incluso en bytes y no generaría ningún problema. También vemos que hay un campo "payload", este campo contendrá toda la información del evento, y por tanto deberá ser la representación del estado de algo. El tipo de este campo es un String o texto, de esta manera pomos incluir tanto JSON o XML, aunque al igual que para el evento y por las misma razones se ha considerado, que para la mayoría de situaciones es más recomendable el uso de JSON. Sea cual sea el formato usado se deberá indicar en la variables "eventDataFormat". Es importante que tanto la estructura del evento como la estructura del payload estén bien documentadas, ya que todos los microservicios, tanto productores como consumidores deberán respetar la misma estructura, de lo contrario nos encontraremos con errores durante el procesamiento de eventos.

En la estructura anterior están los campos mínimos que se han considerado esenciales para un correcto funcionamiento de la arquitectura, sin embargo, existen otros [8] que también aparecerán con bastante frecuencia, dependiendo de las características de nuestro sistema, y son las siguientes:

- *Expiración*: Este parámetro realmente se podría considerar esencial, sin embargo solo lo pondremos cuando los eventos tienen distintas tiempos de expiración, es decir, que un mismo productor genera eventos y estos, en función de alguna característica, le pone un tiempo de expiración u otro. Cuando todos los eventos tienen el mismo tiempo de expiración nos será más fácil y eficiente configurar en el gestor de eventos una política de borrado por tiempo y que sea este quien se encargue de eliminar el evento pasado este tiempo. De esta forma podremos quitar carga al consumidor y reduciremos el número de eventos almacenados en el gestor.
- *Server*: En él indicaremos el servidor en el que se ha generado el evento, como por ejemplo mediante una IP. Este parámetro será especialmente útil si se dan dos situaciones, primero como veremos más adelante EDA puede ayudarnos a generar estadísticas de nuestro sistema, recorriendo la lista de eventos almacenados y viendo los estados en cada momento y como segunda situación cuando un mismo productor de eventos este en más de un servidor, como podría ser por un cluster.
- *UpdateDate*: Este parámetro se relaciona de manera directa con lo mencionado anteriormente de la extensión de eventos tras su creación.
- *Ubicación*: Aquí indicaremos el lugar de donde viene el evento, como podría ser, el código de un país, municipio, continente o lo que veamos más oportuno. Se utilizara principalmente cuando los eventos sean generados por usuarios en lugar de un servidor, y estos puedan estar en distintas partes del mundo; o cuando sean generados por sensores repartidos por un territorio.

Estos 3 últimos parámetros: Server, UpdateDate y Ubicación, son especialmente útiles en la creación de estadísticas y de auditoría de nuestro sistema, por esta razón no se han considerado como esenciales y solo se añadirán en casos que realmente queramos hacer uso de esta característica de EDA.

Productor

Como su propio nombre indica es el encargado de generar los eventos de nuestro sistema y enviarlos al gestor de eventos correspondiente. El productor deberá generar estos eventos al detectar un cambio de estado en el sistema, en un sensor, como respuesta a una petición o incluso tras procesar otro evento. Cabe mencionar que un mismo productor puede generar varios tipos de eventos, tantos como decida el desarrollador, Esto puede ayudar a la eficiencia en la carga de los consumidores, ya que nos permitirá dividir la información que se quiera transmitir en distintos tipos de evento si se considera oportuno, esto que acabo de mencionar se entiende mejor con un ejemplo: Pongamos que tenemos un mismo sensor que mide temperatura y humedad, tendríamos dos opciones, al detectar un cambio en uno de los dos parámetros enviar un evento con toda la información del sensor, es decir, temperatura y humedad o enviar dos eventos distintos, uno para cada parámetro. Al hacer la separación conseguiremos que los consumidores puedan consumir tan solo la información necesaría y así reducir su carga de trabajo.

Los desarrolladores de los productores deberán ser los encargados de mantener actualizada la documentación de la estructura de los eventos mencionada anteriormente, esta documentación cobrará una gran importancia y es un punto clave de esta arquitectura.

Para los productores, la principal configuración que deberán tener en cuenta los desarrolladores será el control de si los mensajes han sido o no recibidos por el gestor de eventos, existen varios niveles, que se deberán elegir en función de la importancia del evento lanzado, y desarrollar una respuesta para cuando cuando tengamos una respuesta fallida. Los niveles que nos ofrecen son: el evento ha sido recibido, el evento ha sido guardado por parte del líder o si ha sido guardado por todas las réplicas del broker. Para cada nivel de los anteriores el tiempo de respuesta será mayor por parte del gestor.

Consumidor

Llamaremos consumidores a los servicios donde llegan los eventos, y se encargarán de procesarlos y generar una respuesta frente a ellos, esta respuesta puede ser de muchos tipos, ya sea inserción en base de datos, escribir en fichero de logs, procesar la información o generar nuevos eventos y enviarlos de nuevo al broker de eventos. Este último punto es importante, ya que nos esta indicando que un consumidor puede actuar a la vez como consumidor y de productor, es más, puede ser productor de múltiples tipos de eventos y consumidor de distintos tipos de eventos.

Es importante mencionar que en EDA los consumidores no conocen nada de los productores, es decir, ellos solo deben de procesar los eventos sin importar de donde vengan y cuando hayan sido producidos, de esta manera conseguiremos una abstracción total entre microservicios, aunque esta es una característica de la que hablaremos más adelante.

En entornos reales es posible que nos encontremos con servicios duplicados(Cluster) para ofrecer servicios de alta disponibilidad, es decir, resistencia a caídas o trabajar de manera paralela, en estos casos en EDA nos encontraremos los consumidores por duplicado, por lo que deberemos tenerlo en cuenta, de lo contrario es posible que dupliquemos procesos con el tiempo de procesado que ello implique o dobles inserciones en base de datos, dependiendo de lo que hagan los consumidores. Con el fin de evitar esto deberemos crear grupos de consumidores, de esta manera el gestor de eventos mantendrá el control a nivel de grupo de consumidores en lugar de a nivel de un único consumidor.

Gestor de eventos

El gestor de eventos es una de las piezas claves de EDA, será el encargado de recibir los eventos de los productores y hacérselos llegar a los consumidores. Este gestor de eventos podrá estar en el mismo servidor que los servicios o en uno independiente para mejorar su fiabilidad, debemos tener en cuenta que es la pieza en torno a la cual gira nuestro sistema, por como es EDA el sistema puede funcionar si cae un productor o un consumidor, sin embargo la caída del gestor de eventos acabará posiblemente en error y pérdida de información. Por estos motivos es importante que los desarrolladores midan las capacidades de su gestor y lo tengan desplegado como un cluster para asegurar su funcionamiento durante un 100 %, además de elegir el que mejor convenga según las necesidades de su sistema. En el mercado encontramos diversos brokers de mensajería, como son kafka, RabbitMq, OpenAMQ, entre otros; en este capítulo solo tenemos el objetivo de entender lo que es EDA, así que analizaremos algunos brokers y sus ventajas y desventajas en el siguiente capítulo.

A parte de encontrar el mejor gestor de eventos para nuestro sistema, hay otro factor que es todavía más importante, que es la configuración del gestor de eventos. Aquí encontramos dos niveles de configuración, uno a nivel de hardware y , a nivel de software. A nivel de hardware, como he dicho anteriormente es importante que el gestor de eventos este siempre disponible, por lo que deberemos decidir la manera de desplegarlo: individual o cluster; en la misma máquina que el resto de servicios o en máquinas distintas. Como es lógico siempre que se pueda es recomendable montar un cluster frente a un gestor individual para estar protegido frente a caídas del servicio. Por el otro lado el separarlo en máquinas o tenerlo todo en uno, es una cuestión más del presupuesto del proyecto, en caso de disponer de varias máquinas sera recomendable mantenerlo separado, así evitaremos que le gestor de eventos herede problemas de rendimiento, CPU o RAM, que puedan ocasionar el resto de servicios de la máquina, de tomar la decisión de separarlo, deberemos tener en cuenta la seguridad de comunicación entre ambas máquinas.

A nivel de software encontramos 3 configuraciones que se han considerado las más importantes para la arquitectura, que son: retención de los mensajes, política de limpieza y política de compresión.

- *Retención* Esta configuración indicará cuanto tiempo se mantendrán disponibles para ser consumidos por parte de los consumidores. Esta configuración puede ser por tiempo o por tamaño máximo.
- *Política de limpieza* Esta nos indicará que hacer con los eventos una vez que se ha cumplido la política de retención, es decir, se ha pasado el tiempo máximo de un evento o se sobrepasado el tamaño máximo. Podemos configurarlo para que el gestor de eventos elimine los mensajes, que es la opción por defecto; para que compacte los mensajes, lo que hará que nos los perdamos y pesen menos en disco; por último encontramos la opción de realizar las dos anteriores, lo que hará que si se cumple la retención por tiempo los mensajes sean compactados y si se cumple la retención por espacio serán eliminados por completo del sistema.
- *Compresión* La compresión de mensajes nos permite guardar los mensajes comprimidos, de esta manera ocuparán menos tamaño y la comunicación será mayor. Realmente esta es una configuración global de la arquitectura, ya que tendrá que estar presente tanto en el productor como en el consumidor.

3

Análisis EDA

Ahora que ya conocemos en que consiste EDA y las distintas partes que lo componen vamos a realizar el análisis como tal de la arquitectura. Para ello comenzaremos sacando las limitaciones que tiene una arquitectura tradicional, como es la basada en llamadas REST, se compara con esta por ser a dia de hoy el principal método de comunicación entre microservicios. Una vez sacadas estas limitaciones veremos si EDA nos ayuda a solventarlas. También veremos el resto de ventajas que nos ofrece esta arquitectura, y se mostrará el principal problema que tiene la implantación de esta arquitectura para los desarrolladores, que es el control de errores.

3.1 Limitaciones actuales

Como se ha mencionado en la introducción de este capítulo vamos a tratar como sistema actual a la arquitectura basada en llamadas REST, es decir, que sigue un esquema petición-respuesta, debido a que es la arquitectura más usada por parte de las empresas informáticas en la actualidad.

A pesar de que esta arquitectura tradicional tiene una gran cantidad de ventajas, como la facilidad de implementación, la eficiencia, seguridad, etc... También nos encontramos ciertas limitaciones, que serám expuestas y explicadas a continuación.

Síncrono

Las arquitecturas basadas en el esquema petición-respuesta son síncronas, esto quiere decir que el servicio que realiza la petición debe esperar a que el servicio destino la procese y genere una respuesta. Por norma general estas respuestas son inmediatas y no generarán ningún problema, pero existen ocasiones que los servicios tardan en procesar la petición, ya sea por una operación de búsqueda de mucha información, poca capacidad de procesamiento, muchos usuarios simultáneos, etc... Durante el tiempo de procesamiento de esta petición el emisor de la petición queda bloqueado hasta recibir una respuesta.

Disponibilidad

Esta limitación está muy unida con la anterior, y es que todos los servicios deben estar disponibles en todo momento para procesar peticiones. Esto quiere decir que se debe tener mucho cuidado con las caídas de servidores, ya que la caída de uno de los servicios puede producir un error en cadena y que todo el sistema global acabe produciendo un error. Esto puede parecer obvio, y por supuesto un servicio nunca debería estar caído, pero como veremos más adelante EDA nos permite continuar el funcionamiento frente a caídas de servidores sin casi ningún problema.

Dependencia

Como sabemos una de las características de las arquitecturas basadas en microservicios es que cada uno de ellos debe ser lo más independiente posible. Sin embargo con una arquitectura de comunicación tradicional, nos encontramos dependencias directas, como son la de funcionamiento, como hemos visto en los dos puntos anteriores y otra de conocimiento. Lo que se quiere decir con esta segunda dependencia es que los servicios deben conocer toda la arquitectura del sistema, es decir, que servicios hay, dónde se encuentran(IP, nombre del servidor,...) y el protocolo para comunicarnos con ellos.

Errores

En estas arquitecturas se deberá tener cuidado y bien controlado el tema de los errores. Pueden venir dados por dos partes, por un lado, los que reciben la petición, ya que generalmente se reciben parámetros o un payload con la información que deberá ser procesada, por lo tanto los desarrolladores deberán confirmar que estos parámetros o payload sean del tipo adecuado, o tengan la estructura adecuada, y devolver el código de respuesta adecuado para evitar problemas. De no ser así es posible que se introduzcan datos corruptos o se produzcan errores internos más difíciles de detectar durante la ejecución.

Por otro lado también deberemos controlar la información que nos llega tras una petición, es decir, en la respuesta, tanto que los códigos de respuesta sean los esperados como comprobar que el contenido recibido tenga una estructura correcta con lo esperado.

Es posible que existan muchas más limitaciones de estas arquitecturas, pero considero que están son las más importante y por tanto las que deberemos tener en cuenta para el análisis. En la siguiente sección se van a mostrar las ventajas y características de EDA y de esta manera ver si nos elimina alguna de las limitaciones anteriores.

3.2 Ventajas EDA

En esta sección veremos las principales ventajas que se han encontrado en el uso de EDA frente a otras alternativas tradicionales como puede ser REST.

Independencia

En EDA los consumidores y los productores no hace falta que se conozcan. Los productores generarán los eventos de manera periódica ya sea por un sensor, cada cierto tiempo, o en reacción a algo, y este será enviado al gestor de eventos. Por lo que estos no tendrán conocimiento, ni necesitan tenerlo, sobre los consumidores. Lo mismo pasa con los consumidores, a ellos les llegará un evento y tendrán que procesarlo, sin importar de dónde venga, quién lo produzca y cuándo haya sido producido(será tarea del gestor de eventos eliminar mensajes antiguos según nos

convenga). Esto implica que tampoco se necesite saber las direcciones dónde están desplegados los microservicios, así conseguimos que si necesitamos cambiar un microservicio de servidor o el path de acceso, no tengamos que realizar ninguna modificación extra en el resto de servicios. Con todo esto conseguimos un nivel muy alto de independencia, aunque no llegaremos a su totalidad, debido a que los consumidores si deberán conocer las estructuras de los eventos y por supuesto los distintos tipos de eventos existentes y a los que se deben suscribir.

Asíncrono

Los sistemas síncronos nos ofrecen ciertas ventajas como son la inmediatez, es decir, se realiza una petición y directamente obtenemos la información solicitada, por lo que aparentemente parece mejor un sistema con comunicación síncrono frente a uno asíncrono, sin embargo los sistemas asíncronos nos ofrecen una mayor seguridad y protección frente a errores [9] o posibles caídas en los servicios. Esto se debe a que en un sistema síncrono todos los servicios deben estar siempre disponibles para conseguir un correcto funcionamiento. En EDA esto no será asi, ya que los eventos se mantienen en el gestor de eventos encolados hasta que el consumidor esta disponible para procesarlo. Con disponibilidad no solo se entiende que un microservicio no este caído, ya que por la naturaleza del mismo puede ser que este en correcto funcionamiento, pero sin embargo se encuentre realizando algún proceso y por lo tanto bloqueado. En resumen, gracias a ser EDA una arquitectura asíncrona, nos ayuda a solucionar problemas de disponibilidad y pérdidas de información, ya que los eventos permanecerán guardados en el gestor.

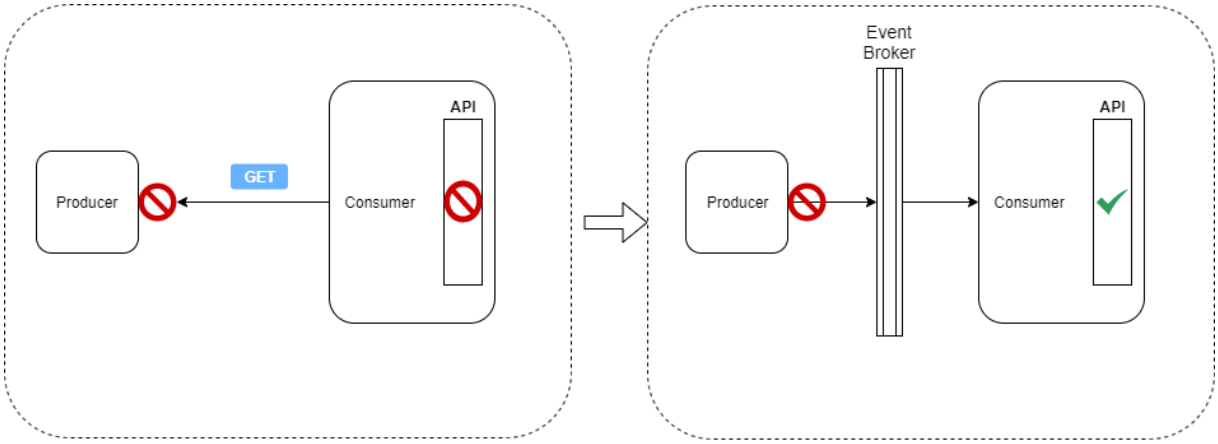

Figura 3.1: Disponibilidad REST vs EDA
Fuente: Propia [10]

En la figura 3.1 podemos ver lo que ocurre frente a una falta de disponibilidad de un servicio que genera la información, en la arquitectura tradicional por REST(izquierda) vemos que el bloqueo del productor, que es el que contiene la información, provoca también el bloqueo de el consumidor, que es el que contiene una API de acceso a la información, sin embargo en EDA(derecha) a pesar de que el productor no este disponible, el consumidor tendrá toda la información necesaria(por lo menos hasta la caída del productor), ya que en un correcto funcionamiento habrá ido recibiendo la información a través de los eventos.

Extensión

En ocasiones se dará el caso de que tengamos que añadir funcionalidades a nuestro sistema, y en una arquitectura basada en microservicios, por facilidad y comodidad será añadiendo un

nuevo microservicio. Este proceso en EDA es muy sencillo gracias a la independencia de la que hemos hablado antes, ya que el procedimiento será tan sencillo como implementar el servicio y suscribirlo a los eventos que necesite según la información que requiera. El resto de servicios no necesitarán ningún tipo de adaptación, como es posible que se necesitaran en una arquitectura tradicional. En la que las comunicaciones están a nivel de código, y tendríamos que modificarlos para poder realizar las peticiones correspondientes.

En la figura 3.1 podemos ver la facilidad de la que hablaba para añadir un nuevo consumidor al sistema, sin realizar ningún cambio de código en el productor, ni gestor de eventos.

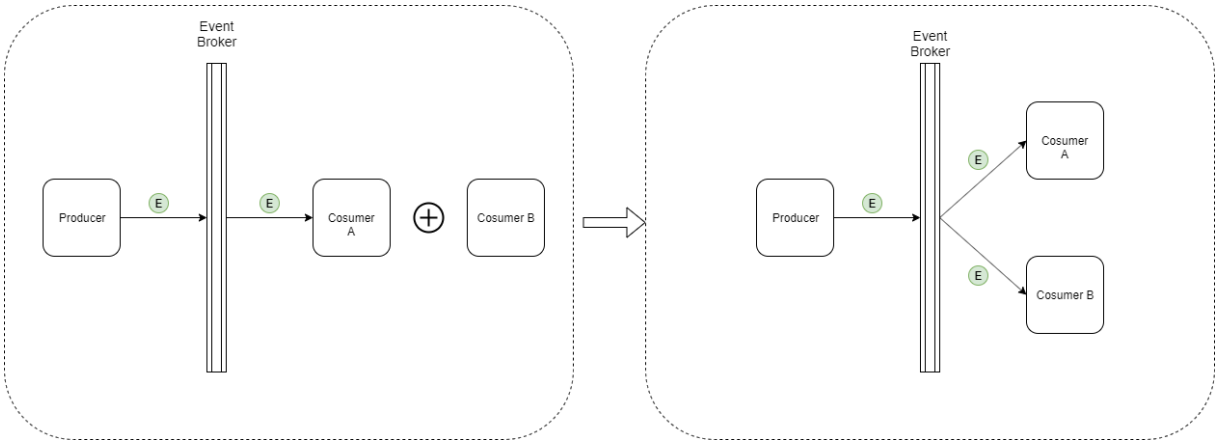

Figura 3.2: Extensión en EDA
Fuente: Propia [10]

Control de estado

De manera más o menos automática e indirecta EDA nos da la posibilidad de tener un control del estado de un servicio o sensor a lo largo del tiempo, esto se debe a que como mencionamos en el punto 2.2.2 un evento se genera por un cambio de estado de "algo", de esta manera, si obtenemos toda la lista de eventos y los procesamos podremos ver los estados por los que ha pasado el sistema en cada momento. Si mantenemos una estructura similar para todos los tipos de eventos del sistema, podríamos tener un servicio que analice todos los tipos de una manera sencilla y rápida. Más adelante, se mostrará un módulo que se ha implementado y que, entre otras cosas, es capaz de generar dashboards a partir de la información que mantiene guardado el gestor de eventos.

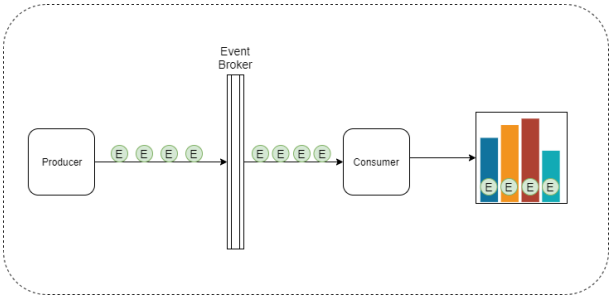

Figura 3.3: Generación de estadísticas en EDA
Fuente: Propia [10]

Aquí deberemos tener en cuenta la configuración de nuestro gestor, como por ejemplo, tiempo que se mantienen los mensajes guardados o las políticas de compresión y limpieza usadas, ya que puede darse el caso de que debido a ellas los datos obtenidos no sean una imagen real de lo sucedido en el periodo de tiempo.

Réplicas

Una de las características de las arquitecturas basadas en microservicios es la independencia de sus BBDD ya que cada microservicio deberá guardar su propia información. En ocasiones para optimizar velocidad los microservicios pueden tener copias locales de parte de las BBDD de otros microservicios que usaran como una memoria caché local. El mantener actualizadas estas réplicas en los sistemas tradicionales puede ser altamente ineficiente por la cantidad de peticiones enviadas. Sin embargo EDA nos facilita el trabajo y la optimización, ya que al realizar una actualización, el servicio envía un evento con la información, en un sistema tradicional habría que enviar tantas como réplicas, y el resto de servicios que se quiera procesarán dicho evento, cuando estén disponibles, y realizarán la inserción en un BBDD local. Obviamente, esta réplica no debe ser igual, sino que se procesará el evento, se obtendrá la información que necesite e insertará solo la parte necesaria.

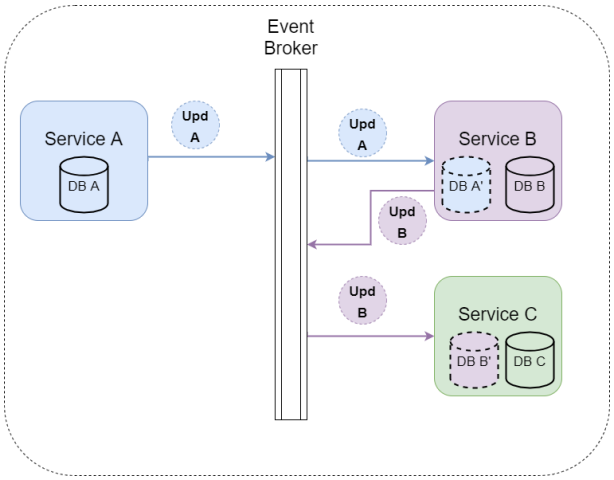

Figura 3.4: Réplica de BBDD mediante EDA
Fuente: Propia [10]

En la figura 3.4 podemos ver el esquema del funcionamiento del mantenimiento de estas réplicas mediante el intercambio de eventos entre servicios.

Recuperación

En EDA todos los eventos generados son almacenados en nuestro gestor de eventos, un mayor o menor tiempo en función de la configuración. Esto hace posible una recuperación [11] de datos en caso de pérdida de información por cualquier motivo. Esto lo obtendremos de manera casi automática ya que los desarrolladores solo deberán implementar un sistema de lectura de los eventos y relanzarlos tal cual al gestor, de esta manera volverán a ser procesados

3.3 Limitaciones y desventajas EDA

En la sección anterior se han descrito las que se han considerado como principales ventajas de EDA en la comunicación entre microservicios, sin embargo, EDA no es un arquitectura perfecta y tiene ciertas limitaciones y desventajas, con las cuales se encontrarán los desarrolladores a la hora de su implantación y tendrán que invertir un tiempo extra para evitar problemas mayores durante la ejecución.

Control de errores

Una de las principales desventajas que tiene esta arquitectura y en la que más tiempo tendrán que invertir los desarrolladores y arquitectos es en el control de errores, esto se debe justamente a la independencia y la unidireccionalidad del flujo de eventos, que provoca una difícil reacción frente a posible errores. Encontramos dos posibles grupos de errores:

- Errores del *Gestor de eventos*. Son los más críticos que nos encontraremos por girar toda la arquitectura entorno a él, aunque deberían de aparecer menos ya que como hemos comentado anteriormente en el punto 2.2.2, se deberán tomar medidas anteriores para evitarlos, ya que principalmente son generados por falta de disponibilidad del gestor.
- Errores en los *Microservicios*. Tanto por parte de los productores, durante la generación o envío; o por parte de los consumidores, durante el procesado del evento o la inserción de la información en BBDD. Estos errores normalmente serán menos críticos que los anteriores y más fáciles de controlar. Además por norma general en EDA se mueven una gran cantidad de eventos, y la pérdida de un pequeño porcentaje es probable que no afecte a los resultados finales, esto por supuesto dependerá totalmente del caso concreto de cada uno.

Los errores en el **Gestor de eventos** se dan principalmente porque este no esta disponible, por lo que los desarrolladores deberán implementar un sistema alternativo para no perder toda la información. Una posible solución sera guardar de manera temporal y local estos eventos en el propio servicio a la espera de que el gestor se recupere, una vez recuperado enviará todos de golpe y los borrará del sistema local, es importante que para esto tengamos el evento guardado tal cual se genero en su momento, incluida la marca de tiempo, recordamos que un evento es como una imagen del estado de algo, por lo que para el procesado no debería afectar que llegue más tarde, ya que en su interior estará la fecha original de creación. Destacar que con esto el sistema continuará funcionando durante el error del gestor, sin embargo información de los consumidores no estará actualizada durante este periodo.

Lo más importante para los desarrolladores y arquitectos sin embargo es evitar este tipo de problemas, con un análisis previo, eligiendo el gestor de eventos que más convenga y principalmente asignarle los recursos que vaya a necesitar, así como, mantener duplicados los gestores en forma de cluster.

Los errores en los **Microservicios productores** pueden ser durante el envío, es decir, no encontrar el gestor de eventos: por un problema de conexión o de seguridad, y este caso lo trataremos como si fuera un error en el gestor y podremos hacer lo dicho anteriormente. Otro tipo de error puede ser durante la generación del mismo. Aquí el desarrollador deberá decidir si se aborta el proceso de envío, lo que supondrá una pérdida de información o si, por el contrario, sigue el proceso, lo que significará muy probablemente la inserción de datos corruptos en el payload del evento, en este caso, es recomendable indicarlo de alguna forma, como por ejemplo en el

eventName(de la estructura recomendada en la sección 2.2.2) y de esta manera, los consumidores podrán tenerlo en cuenta y realizar un procesado alternativo y evitar errores posteriores.

Desborde

EDA contempla un problema heredado de cómo y cuándo se generan los eventos(tras un cambio de estado). Estos cambios de estado en el sistema puede ser de muchos tipos, algunos predecibles como puede ser lanzar un evento a la llegada una hora determinada, pero otros no son predecibles, como puede ser un sensor. En este último caso se puede hacer un estimación aproximada, pero no exacta, y es aquí donde surge el problema, y es que se pueden generar una gran cantidad de eventos por parte de los productores y que los servicios consumidores no sean capaces de procesarlos lo suficientemente rápido, es cierto, que gracias al gestor de eventos es difícil que estos se pierdan(dependiendo de la política configurada); pero si esto ocurre de manera continuada ira provocando que la información procesada final, no este actualizada. En un sistema tradicional al ser síncrono esto es más difícil que pase, ya que el productor no podrá enviar la información a procesar hasta que el consumidor haya procesado la petición anterior.

Cambios en los eventos

Se ha mencionado en varias ocasiones que la principal ventaja que nos ofrece EDA es el alto grado de independencia y desacople entre microservicios, debido a que no deben conocerse entre ellos, ya que el gestor es el que se encargará de transmitir la información entre ellos. Sin embargo, si encontramos un aspecto que los consumidores deben conocer de los productores, y es la estructura del evento, siendo este aspecto el único grado de dependencia que nos genera entre servicios.

Esto supone que un cambio en el modelo de evento del productor provoque que se tengan que modificar todas las estructuras de los consumidores, además, este cambio debe hacerse de manera inmediata, ya que mientras las estructuras no coincidan, como es obvio, los consumidores estarán en un estado de error.

4

Casos de uso

Para finalizar con la parte de análisis y, antes de comenzar, con los módulos desarrollados, he visto necesario realizar un apartado con los casos de uso que mejor cuadran con EDA. Realmente estos casos de uso ya han sido mencionados de manera directa o indirecta a lo largo de los apartados anteriores. Por lo que en este se pretende dejar claro cada uno de ellos y extender la información sobre los mismos.

4.1 Comunicación entre microservicios

Este es el caso de uso más claro y que en mi opinión mejor cuadra con EDA y del cual más se ha estado mencionando anteriormente. Se trata de una comunicación entre microservicios, es decir, sustituir la comunicación api-rest por los eventos y un gestor de eventos. Este caso de uso es el que mejor cuadra debido a que las características de cada uno se complementan perfectamente con las del otro es decir, la característica principal de los microservicios es que estén altamente desacoplados [12]. El acoplamiento es una "medida" del grado de interconexión entre servicios y, por tanto, también nos dice el grado de independencia entre servicios. Esta medida se puede estimar según: el tipo de conexión entre módulos, la cantidad de elementos pasados(que no de datos), el tipo de información que se transfiere y el tiempo de enlace de la conexión. Con EDA podemos conseguir un alto nivel de desacople entre nuestros microservicios debido a que no estamos ante una conexión directa entre ellos, es más, prácticamente no deben conocerse, es por ello que EDA encaja muy bien en una arquitectura basada en microservicios.

Este caso de uso se puede dividir a su vez en tres subcasos, según la funcionalidad que nos aporte cada uno:

4.1.1. Flujo de procesamiento

Lo podríamos definir como el caso base, se trata de una conexión"directa' entre nuestros servicios a través del gestor de eventos. El funcionamiento es simple, el productor detecta un

cambio de estado, ya sea por un sensor, tiempo, error, etc. y lanza el evento con la información al gestor; este en cuanto sea posible se lo mandará a los consumidores correspondientes para su procesado.

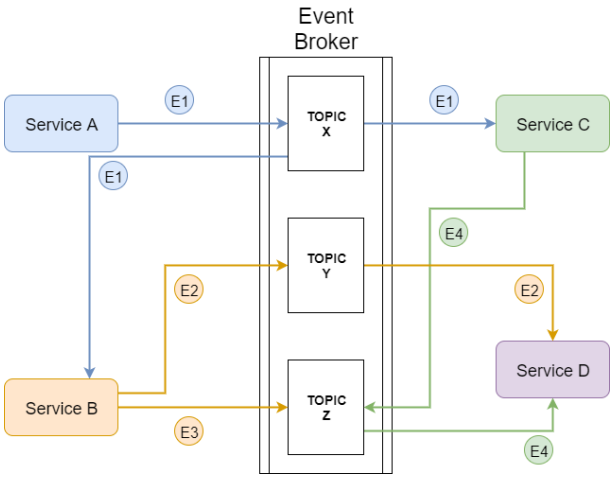

Figura 4.1: Esquema de flujo entre microservicios
Fuente: propia [10]

En la figura 4.1 podemos ver un esquema de la arquitectura de lo que sería este caso de uso y de la propagación de los eventos por el sistema.

4.1.2. Sicronización de datos

La independencia entre microservicios debe incluir también una independencia de sus bases de datos, es decir, que cada microservicio tenga su propia base de datos con su propia información. Sin embargo los datos deben ser accesibles desde diferentes servicios. Esto se puede hacer por peticiones REST, sin embargo generará latencia por el tiempo extra de peticiones. Una solución para esto es crear una réplica local en cada microservicio a modo de memoria caché local. Mantener estas cachés actualizadas via REST puede ser un proceso costoso debido a ser un proceso síncrono, por lo tanto un buen enfoque para afrontarlo es mediante EDA, con la utilización de eventos que se generan cuando se realiza una actualización en bases de datos y los microservicios que lo requieran consumirán estos eventos para mantener sus cachés locales. Es importante mencionar que estas réplicas no deberán ser exactamente iguales que las de los servicios originales ya que romperíamos la independencia de bases de datos. Estas réplicas contendrán solo una parte de la información, para agilizar el procesado; es decir, imaginemos que tenemos un servicio A con con una BD de 10 tablas, podríamos crear una réplica local en un servicio B que contenga tan solo 2 y además aplicar un filtrado anterior a la inserción, así conseguiremos que el servicio solo mantenga en su BD información que realmente vaya a ser útil para él y evitar la mayor redundancia posible.

Como podemos ver, este caso de uso realmente es una derivación del anterior pero con un enfoque distinto en cuanto a la funcionalidad.

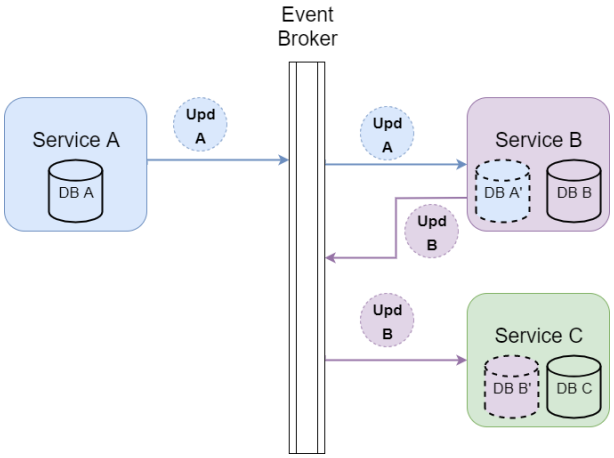

Figura 4.2: Sicronización de datos
Fuente: propia [10]

En el esquema 4.2 podemos ver como seria el flujo de eventos para mantener estas replicas.

4.1.3. Event Sourcing

El Event Sourcing es una manera distinta en cuanto al consumo de eventos. Consiste en ir guardando los eventos, que contendrán información del estado del sistema y se irá generando un registro, de esta forma, en lugar de que los consumidores vaya consumiendo los eventos según se generen, estos harán una lectura de todo el registro con un posible filtrado.

4.2 APIs dirigidas por eventos

En la actualidad, una gran cantidad de empresas y organizaciones [13] hacen uso de APIs web, públicas y privadas, para ofrecer una manera de obtener información a sus clientes. Estas son del tipo REST, es decir, mediante una petición http contra el endpoint deseado obtenemos la información requerida.

La limitación que tienen actualmente estos sistemas es que son los usuarios o clientes los que deben solicitar la información al servidor, por ello, para mantener datos actualizados en directo se deberán realizar una gran cantidad de llamadas a los servicios. EDA nos ofrece la solución a esta limitación, ya que no serán los consumidores los que soliciten la información, sino que serán los productores los que la envíen una vez detecten un cambio de estado. De esta manera, se obtendrá la información justo en el momento del cambio de estado y la podrán mantener actualizada sin la necesidad de estar realizando las llamadas continuamente. Gracias a esto reduciremos en gran medida el tráfico y por tanto mejoraremos el rendimiento de nuestro servidor.

Este es uno de esos casos en los que EDA no viene a sustituir a REST, sino a convivir. Nos encontraremos situaciones en las que EDA nos dará un mejor resultado, por ejemplo, cuando estamos ante sensores, o cosas que cambian de estado, el seguimiento de un avión; y nos encontraremos otras situaciones en las que directamente EDA no podrá ser aplicable, como por ejemplo APIs para obtener información de una BD. Además a día de hoy estas APIs REST tienen un alto grado de estandarización por el gran número de usuarios que tienen, por lo que para

las organizaciones es más cómodo y económico utilizar el enfoque tradicional.Esto es probable que vaya cambiando, debido a que cada vez se esta hablando más de esta arquitectura y tomando un mayor interés por parte de las empresas.

4.3 Ingesta de datos

EDA también nos permite el consumo de un flujo constante de eventos en lugar de un consumo individual. Será parecido a lo ya visto de productor consumidor entre microservicios, con la diferencia que en este caso el microservicio consumidor lo sustituiremos por un software de ingesta y analítica de datos, como podría ser "StreamSets Data Collector" [14], que es el que se ha utilizado para realizar las pruebas ya que viene incluido en las tecnologías de la OP.

Estos programas nos permiten configurar una fuente de datos, que en el caso de EDA sería el broker de eventos y, a continuación, los distintos procesos que iremos aplicando a cada evento, hasta llegar al final del flujo, que indicaremos el destino de la información ya procesada, que podría ser una inserción a DB, logs o generar un nuevo evento. En la siguiente figura podemos ver el esquema que representa este caso:

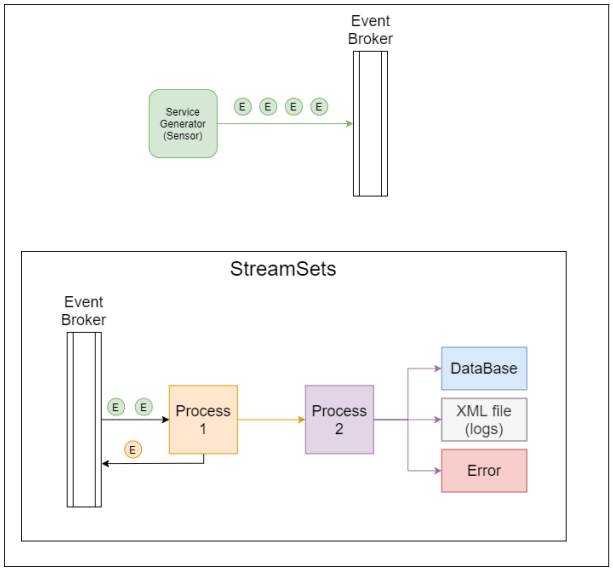

Figura 4.3: Procesamiento flujo de eventos
Fuente: propia [10]

5 Ejemplo EDA

En este punto ya se da por finalizada la parte de análisis de EDA, y comienza la segunda parte del TFG, que corresponde a los módulos desarrollados. Estos han sido dos, el primero que será el desarrollado en este punto, es un ejemplo con múltiples microservicios, tanto productores como consumidores, conectados entre ellos tan solo por el gestor de eventos. Esto es una exageración ya que como hemos comentado, EDA viene a convivir con otras arquitecturas como puede ser REST. Sin embargo se ha querido llevar al extremo para ver que es totalmente posible implementar un sistema totalmente funcional con EDA, lo que no quiere decir que sea la mejor opción.

A través de este ejemplo, también se busca dar pautas de cómo se realizaría una implementación de EDA en un sistema: configuración de los servicios, configuración de kafka, gestion de errores, control de las estructuras de los eventos, etc. Este ejemplo se suministrará junto con el análisis más teórico para que sirva de referencia para su implementación en distintos productos de la compañía.

5.1 Introducción

Como hemos mencionado este primer módulo desarrollado va a ser un ejemplo de EDA. En él vamos a realizar una simulación de un aeropuerto en el cual se van a generar un conjunto de aviones virtuales, tantos como se quiera, y estos irán pasando por distintos estados como pueden ser: “en ruta”, “salida” o “retraso”, entre otros. Cada vez que un avión realice un cambio de estado, este será enviado al gestor de eventos. Estos eventos serán recogidos por los distintos consumidores y en función del tipo reaccionarán a ellos de distintas formas, asignando puertas de embarque, publicando la información de un panel o asignando pista al vuelo. En las siguientes secciones veremos esto más en detalle. Vemos, por tanto, como estamos ante un caso en el que el productor no va a ser un único servicio sino que tendremos un conjunto de productores, tantos como aviones.

5.2 Arquitectura

En esta sección vamos a ver las distintas partes que forman el conjunto del ejemplo y cómo se realiza la conexión entre ellas. Primero de una forma más general con un esquema de toda la arquitectura y después pasaremos a detallar el funcionamiento de cada uno de los distintos módulos.

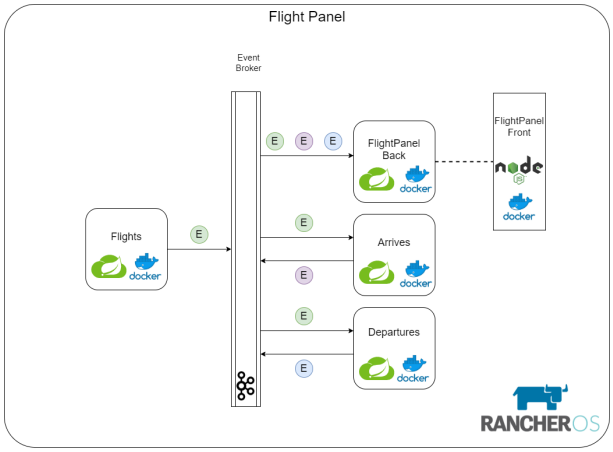

Figura 5.1: Arquitectura ejemplo aeropuerto

En el esquema 5.1 tenemos la representación de la arquitectura en la cual podemos ver 4 microservicios backend y uno frontend. Lo más importante y que cabe mencionar, como no podía ser de otro modo, es la conexión que existe entre estos servicios(backend), ya que no tenemos conexiones directas entre ellos sino que utilizamos nuestro gestor de eventos(kafka) como intermediario. También podemos ver los distintos tipos de eventos a través de los colores, que cada uno ira a un topic de kafka distinto. A continuación, veremos el funcionamiento de cada módulo:

5.2.1. Microservicio “Flights”

Este microservicio es el simulador como tal, su funcionamiento de forma general es bastante sencillo, mediante una API REST el usuario podrá realizar un inicio con una petición GET pasandole la cantidad de vuelos que se deseen generar, al endpoint: “http://<server>:<port>/api/start/N” siendo N un entero indicando el número de vuelos que se deseen. El sistema generará tantos hilos como vuelos se hayan indicado, y a partir de este punto, cada hilo actuará de mánera independiente realizando cambios de estado aleatorios pero siguiendo una lógica lo más real posible para un vuelo, en el esquema 5.2 vemos la representación.

Servicio Gestión Vuelos

Este servicio será el encargado de permitirnos gestionar la simulación de nuestros vuelos, a través de una API. Como ya hemos mencionado se nos permite comenzar la simulación, detenerla y obtener información de la misma: arrancada, en pausa o el número total de eventos generados y de hilos en funcionamiento.

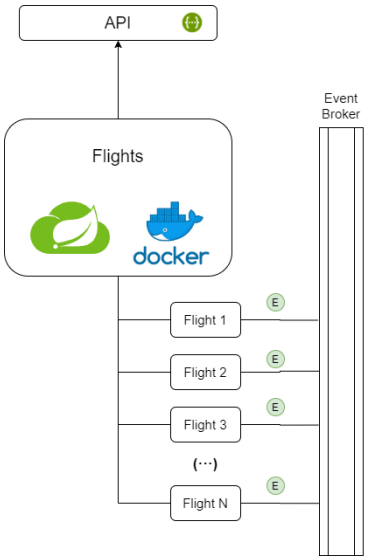

Figura 5.2: Esquema de microservicio de vuelos

```
public void init(int numFlights) {
  flights.clear();
  for(int i=0;i<numFlights; i++) {
    Thread simThread;
    Flight f = new Flight(eda);
    simThread = new Thread(f);
    flights.add(simThread);
  }
}
```

En el método anterior vemos como se realiza la inicialización de cada uno de los hilos correspondientes a un vuelo y se almacena en un array, para mas tarde realizar un "start()" de estos hilos. Al generar cada uno, se le asigna un número aleatorio y un prefijo de aerolínea entre un conjunto determinado. A continuación, cada uno de ellos se establece en uno de los estados disponibles, también de forma aleatoria. Se esta manera se tiene una mayor sensación de realismo. Como podemos ver, a cada vuelo se le pasa también una variable llamada service, este es el servicio que se encargará de realizar la conexión con kafka y permitir el envío de eventos.

Estados Vuelos

A continuación, nos centraremos en cada vuelo. Ya hemos visto que cada vuelo se genera con un número aleatorio que actuará como su identificador y se establece un estado inicial. Después de esto, entraremos en un bucle infinito(hasta que el usuario mande la orden de interrupción al hilo), en este bucle se producirán los cambios de estado que van según lo siguiente: cada cierto tiempo, un numero aleatorio entre un numero mínimo y máximo configurable pueden ocurrir dos cosas, la primera es que no tengamos un cambio de estado, por lo que no mandaremos evento y volveremos a esperar el tiempo. La segunda es que se produzca un cambio de estado, aquí cada estado tendrá fijado los siguientes estados posibles para dar una mayor sensación de realismo. Una vez que se produce este cambio, el sistema notifica del estado actual del sistema al gestor de eventos y volvemos a esperar el tiempo que sea. A continuación, veremos algunos fragmentos

del código que tenemos.

```
private static final String[] PRE_AVION = { "UAL", "IBE", "ACA", "DLH",↩
     "SWR", "AAL", "VLG", "RYR", "AVA", "JAL",
    "AMX", "KLM", "ARG" };
private static final String[] ESTADOS_AVION = { "SALIDA", "LLEGADA", "↩
    RETRASO", "RUTA", "EMBARQUE", "ULT_LLAMADA" };
private static final String[] DESTINOS = { "MADRID", "BARCELONA", "↩
    LONDRES", "NEW YORK", "PARIS", "BERLIN" };
```

En el fragmento superior vemos las declaraciones de los prefijos que puede tomar el número de vuelo durante el inicio, los distintos estados que puede tomar nuestro vuelo y los distintos destino que puede tener.

```
UUID uuid = UUID.randomUUID();
Timestamp fechaEvento= new Timestamp(System.currentTimeMillis());
String json;
try {
  ObjectMapper mapper = new ObjectMapper();
  json = mapper.writeValueAsString(this);
} catch (JsonProcessingException e) {
  log.error("Error sending data");
  return;
}
Event event=new Event(uuid.toString(), "FLIGHT", fechaEvento, this.↩
    numero, this.version, "JSON", json);
eda.send(event, "flights-events");
```

En este bloque vemos una de las partes más importantes: la generación del evento y el envío del mismo. Vemos como realizamos la conversión de Objeto Java a Json con ayuda de la clase ObjectMapper. Para introducir solo lo valores que se desean también se ha hecho uso de la etiqueta "@JsonIgnore" sobre algunas variables como son el "EdaService", con el cual enviamos los eventos o el estado anterior del vuelo, que se usa internamente para realizar el cálculo del próximo evento. Una vez que tenemos el JSON generamos la instancia de nuestra clase Event, que sigue la estructura vista en el punto 2.2.2, como primer argumento pasamos un identificador del evento, en este caso al tratarse de un ejemplo se utiliza un UUID aleatorio; en segundo mensaje el nombre del evento; en tercer lugar la fecha de creación; en cuarto lugar el autor, que en este caso será el numero de vuelo puesto que simulamos que tenemos múltiples productores; en quinto lugar se pasa la versión de la estructura del payload del evento, que es cargada del archivo application.yml gracias a la etiqueta @Value(); en sexto lugar el formato del payload y por último le pasamos nuestro payload generado previamente. Por último pasamos a nuestro servicio, que veremos a continuación parte de su implementación, el evento y un topic al que pasarlo.

```
@Autowired
KafkaTemplate<String,String> kafkaTemplate;
(...)
kafkaTemplate.send(topic,jsonEvent);
```

En este último fragmento de código se ha cogido las dos partes más importantes: son la declaración del *KafkaTemplate*, que es parte de la dependencia de kafka para Spring que se declará mediante la etiqueta @Autowired para que Spring busque de manera automática el Bean correspondiente y además busque la configuración necesaria en nuestro proyecto. En este caso, está presente en el application.yml, que lo más importante es indicar servidor y puerto dónde esté nuestro broker. También vemos que es del tipo "<String, String>" esto es así para mayor comodidad en el código, por debajo kafka lo almacenará en bytes. Una vez que tenemos la instancia será tan fácil como llamar a su método *send()*, que tiene como primer argumento el topic al que se desee enviar y en segundo lugar el evento, como ya se mencionó el formato elegido como mejor opción es JSON, por lo que se transforma el objecto Event y se envía.

5.2.2. Microservicios "Arrives" y "Departures"

Ambos servicios son muy parecidos, aunque cada uno procesa un tipo de evento y generá una respuesta diferente, pero su funcionamiento es idéntico por lo que se ha decidido agruparlos en el mismo punto. Estos servicios son consumidores de los eventos generados por los vuelos vistos anteriormente. Cada uno recibe todos los eventos, los analiza y realiza un filtrado para comprobar si es del tipo requerido. En esto vemos una de las decisiones que se deben tomar durante el diseño, ya que podríamos haber dividido los eventos en más *topics* y de esta forma disminuir la carga en los consumidores ya que no tendrían que realizar el filtrado y les llegarían solo los eventos de vuelos que necesiten. Esta manera es más genérica y debido al bajo volumen de eventos que llegan no supone un problema, pero para un sistema con mayor volumen sería más recomendable dividir en *topics*. A continuación vemos el código correspondiente para generar los consumidores:

```
@KafkaListener(topics = "flights-events", groupId = "flights-departures↩
    ")
public void consumeFlight(String jsonEvent) {
  ObjectMapper mapper = new ObjectMapper();
  try {
    Event event;
    Flight flight;

    event = mapper.readValue(jsonEvent, Event.class);
    flight = mapper.readValue(event.getPayLoad(), Flight.class);
    (...)
```

Como vemos en este fragmento, gracias a Spring, vuelve a ser relativamente fácil generar un consumidor de kafka, esto lo hacemos con la etiqueta @KafkaListener(), a la cual le pasamos dos parámetros: el primero corresponde al topic al cual queremos vincularnos y el segundo el grupo de consumidores al que pertenece. Este grupo es muy importante indicarlo ya que será el encargado de evitar duplicados en nuestros sistema y que los eventos sean cosumidos por duplicado. Esto puede ocurrir principalmente en entornos de alta disponibilidad, que hacen uso de clúster de servicios y sin este grupo ambos servicios podrían consumir el mismo evento a pesar de ser realmente el mismo. Esta etiqueta al igual que pasaba con el productor busca la configuración en el proyecto, y en este caso vuelve a estar en el application.yml. También podemos ver como el objecto que llega es de tipo String y es el JSON del evento, es ahora cuando vemos la importancia de que nuestros servicios compartan las estructuras de los eventos, ya que un simple

cambio provoca que el sistema no sepa intepretarlos y por lo tanto lo rechace.

Una vez que ya se ha parseado el evento, cada uno de los servicios genera su respuesta correspondiente. En el caso de "Arrives" genera una cinta para recoger el equipaje y en el caso de "departures" genera puertas de embarque, con esta información se genera un nuevo evento y este es enviado de vuelta a kafka a sus topics correspondientes. Por lo tanto, estamos ante un caso de servicio que actúa como consumidor y productor de manera simultanea.

5.2.3. Microservicio "FlightPanel Back"

Por último, en cuanto a los servicios backend, tenemos el que se encarga de unir toda la información que se genera por parte de los anteriores y mantenerla ordenada. Esto lo hace convertiendose en consumidor de los 3 topics vistos anteriormente(flights, arrives y departures). Se hace exactamente igual que en lo visto en el apartado 5.2.2, es decir mendiante el @KafkaListener.

Este servicio tiene dos formas de obtener la información, por un lado contiene un endpoint que nos devolvera la lista de vuelos que va recibiendo y que mantiene actualizada. Estos vuelos también tienen presente la cinta para el equipaje y la puerta de embarque, que se pondrán a *null* en los estados en los que no esté presente esta información. Por otro lado, la forma de enviar la información es a traves del módulo frontend, desde este se creará una conexión mediante *Stream Sockets* con el frontal y se utilizará esta conexión para enviar los datos según vayan llegando al servicio.

A continuación se muestra la configuración utilizada para abrir esta conexión:

```
private Map<String, WebSocketSession> sessions = new ↩
    ConcurrentHashMap<>();
(...)
public void sendMessageToAll(String message) {
  TextMessage textMessage = new TextMessage(message);
  sessions.forEach((key, value) -> {
    try {
      value.sendMessage(textMessage);
      log.info("Send message {} to socketId: {}", message, key);
    } catch (IOException e) {
      e.printStackTrace();
    }
  });
}
```

Con esta configuración, tan solo deberemos crear una instancia de la clase WebsocketHandler y llamar al metodo send con nuestro mensaje. Vemos como es otro tipo de EDA, ya que tenemos un productor de eventos(Backend), un gestor de eventos(Stream Sockets) y un consumidor(frontend).

5.2.4. Microservicio "FlightPanel Front"

Es un sencillo módulo frontend desarrollado en *Node.js* e implementado en html y js. Se utiliza Node.js para desplegar el servidor web mediante "Express.js" y de esta manera disponibilizar los documentos estaticos HTML.

La parte de código de este módulo es muy simple, por un lado tenemos el HTML, que contiene tan solo una tabla con cinco columnas, una por cada campo que tiene vuelo: número, destino, estado, puerta y la cinta de maletas. Las filas se iran rellenando según la información que le llegue desde el backend, es este fichero también tenemos los estilos que se van a aplicar a la interfaz, para generar una estetica lo más similar posible a un panel de información de un aeropuerto.

Por otro lado tenemos la parte de JavaScript, cuya funcionalidad principal es abrir el canal de comunicación por el cual llegará la información del servidor backend, esto se hace mediante WebSockets de la siguiente manera:

```
ws = new WebSocket('ws://<server>:<port>/flights');
ws.onmessage = function (data) {
  //update table
  addToUi(data.data);
}
```

Como vemos se utiliza la clase WebSocket() para conectarnos al que habiamos abierto desde nuestro backend, y con cada mensaje que llegue por el *WebSocket* llamaremos a la función "addToUi()", que se encargará de parsear la información que llega en formato JSON y pintarla en la tabla html. A continuación vemos el resultado final:

NUMERO	DESTINO	ESTADO	PUERTA	CINTA MALETAS
AVA6882	BARCELONA	RUTA		
ARG4699	LONDRES	EMBARQUE	6A	
AVA7025	BARCELONA	RETRASO		
JAL1562	LONDRES	EMBARQUE	7A	
SWR5410	BERLIN	LLEGADA		2B
ACA3035	NEW YORK	EMBARQUE	13E	
AMX2531	PARIS	SALIDA		
AAL7115	PARIS	RUTA		
AMX9824	BERLIN	EMBARQUE	8E	
IBE3772	BARCELONA	LLEGADA		3A

Figura 5.3: Resultado final panel de vuelos

En el anexo D.1 podemos ver el listado de todas las tecnologías que se han utilizado en este ejemplo.

6

Módulo administración

Como hemos comentado, una de las ventajas que nos ofrece EDA, es la posibilidad de tener un control de estado de nuestro sistema de una manera bastante sencilla, debido a que en nuestro gestor estarán almacenados los eventos, que corresponden a cambios de estados del mismo. Tan solo tendremos que recorrer este conjunto de eventos y podremos ver el estado del sistema en cada momento u obtener distintas estadísticas del sistema.

La generación de estas estadísticas es lo que podremos obtener con este módulo implementado junto con otro tipo información de nuestro gestor de eventos, como número de consumidores o brokers disponibles. Este módulo sera un microservicio implementado en Java mediante Spring Boot y hará uso de la OP, tanto para kafka como para la seguridad del microservicio, que será delegada en el identity manager de la misma. A continuación muestro un esquema de las tecnologías usadas tanto por parte de la OP como del microservicio.

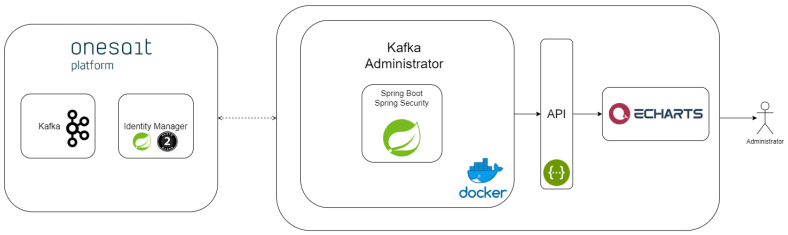

Figura 6.1: kafka administrator architecture
Fuente: propia

En la figura 6.1 encontramos dos bloques: el primero(izquierda) representa la OP con los módulos que se usan de manera directa, que son kafka por un lado, que sera usado como gestor de eventos a monitorizar y el identity manager junto con Oauth2 por otro lado. En ellos se delegará la seguridad de nuestro microservicio. El otro bloque podemos representa el microservicio implementado, que como podemos ver hace uso de *Spring Boot* y *Spring Security*, swagger como

interfaz de la API, eCharts para poder mostrar de manera visual la información al usuario y por último, hará uso de Docker. Este módulo se pondrá a disposición por parte del departamento de arquitectura como un módulo listo para generar una imagen Docker y desplegarse en las maquinas del departamento.

6.1 Funcionalidades

El módulo internamente lo podemos dividir en 3 subsistemas: APIController, KafkaService, eChartsService. Ahora pasaremos a describir cada uno de ellos con más detalle, aunque antes cabe destacar que el APIController y el eChartsService, son ejemplos concretos; mientras que el kafkaService es el módulo que será fijo para el departamento.

Subsistema APIController

Este subsistema se encargará de generar el endpoint para el módulo, es decir, será la puerta de acceso para los usuarios a la información de nuestro módulo. Esta conexión se hará mediante REST y se implementará mediante anotaciones de Spring, exactamente con "@RestController", que indica que nuestra clase Java deberá ser tratada como un controlador REST; y con la etiqueta "@GetMapping("/endpoint")", esta se pondrá encima de los métodos y los expondrá al público. La respuesta de estos métodos será del tipo "ResponseEntity<Object>", que es una respuesta HTTP a la que le podemos indicar distintos headers, el status y el propio contenido, en este caso, el contenido debe ser del tipo Object de Java, para mayor comodidad, aunque podríamos indicar el tipo exacto de nuestra respuesta. A continuación muestro un ejemplo del código de un endpoint.

```
    @GetMapping("/listTopics")
  public ResponseEntity<Object> getTopics() {
    Collection<String> response;
    try {
      response = admin.getTopics();
      return new ResponseEntity<>(response, HttpStatus.OK);
    }catch(KafkaAdminException e) {
      return new ResponseEntity<>(e.getCustomReason(),HttpStatus.↩
          INTERNAL_SERVER_ERROR);
    }
    }
```

En el bloque de código anterior podemos ver que consiste en un endpoint que nos devolverá la lista de topics de nuestro kafka. Esto lo hará mediante una llamada al servicio de administración de kafka(subsistema KafkaService) que explicaremos más adelante. Aunque no parezca una posible respuesta a simple vista, el resultado a este endponit puede ser un error 401, que como se indica en el RFC [15] significa:"The request requires user authentication.". Esto se debe a que es un módulo crítico en cuanto a información por lo que esta protegido mediante SpringSecurity. Que se encarga de manera más o menos automática de devolver este error.

A continuación entremos en los detalles de su configuración.

Seguridad

Puesto que este módulo servirá para darnos estadísticas de nuestro sistema y de nuestro broker de eventos, no puede estar disponible para todo el mundo y se deberá poder configurar para que solo los usuarios con un cierto rol o con una autorización puedan acceder a él. Para ello hemos usado el Identity Manager de la OP, que mediante Oauth2 nos suministra los tokens necesarios. Puesto que los posibles productos que vayan a usar este módulo, por políticas internas, deberán hacer uso de la OP se ha considerado que delegar la autenticación en ella es la mejor opción como método por defecto.

Para realizar esta delegación, se ha hecho uso de "Spring Security" y una librería que nos proveen desde el departamento de plataforma para facilitar el proceso. Con esta librería en nuestro proyecto tan solo deberemos configurar unos parámetros en el application.yml que podemos ver en el anexo B.1, mediante los cuales se creara la conexión. La plataforma hace uso de Oauth2 y tokens JWT para todo lo relacionado con seguridad.

En segundo lugar, una vez que tenemos configurada la conexión con nuestra plataforma, deberemos configurar SpringSecurity para indicar qué usuarios tienen acceso y a dónde. Lo haremos a través de la configuración que encontramos en el segundo bloque de código del anexo B.1. Para dar una seguridad por defecto y que actúe como base, se ha optado por permitir entrar al swagger a todos los usuarios, mediante el método "permitAll()", y que a todos los demás endpoints solo puedan acceder usuarios con rol "ROLE_USER". Es aquí donde en función de cada producto se irán definiendo los distintos niveles de seguridad deseados. También podemos ver la etiqueta "@Profile()", está etiqueta indica que esta clase de configuración solo se utilice si tenemos el perfil de spring indicado, gracias a esto podemos tener dos configuraciones de seguridad, lo cual facilitará el trabajo a los desarrolladores, ya que se podrá configurar que en local no nos pida ningún token de usuario, pero al desplegar el módulo conservar la configuración de seguridad deseada.

La API está documentada mediante swagger2, que además nos ofrece una interfaz para poder realizar peticiones de una manera sencilla a nuestros endpoints, en el Anexo C se puede encontrar más información sobre esta tecnología.

Subsistema KafkaService

Este subsistema es el que se encarga de sacar la información del broker de kafka. Para ello se hace uso de la clase Java AdminClient presente org.apache.kafka.clients.admin.AdminClient, esta clase nos permite obtener bastante información interesante de nuestro gestor de eventos. En este caso se ha usado para obtener el listado de consumidores del sistema, obtener un listado de topics y obtener los brokers disponibles en caso de tener un clúster, algo muy recomenable en sistemas en producción que hagan uso de EDA. Su utilización es muy sencilla, como podemos ver en el siguiente fragemento de código:

```
Properties properties = new Properties();
properties.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, kafkaServer)↩
    ;
AdminClient adminClient = AdminClient.create(properties);
```

Vemos como para la creación de la instancia de la clase, tan solo deberemos indicar el servidor de kafka en las properties y ya estará listo para llamar a sus métodos, En este caso, los métodos que se van a utilizar son: listTopics(), listConsumerGroups() y describeCluster(). A pesar de que en este caso solo utilice estas, en la documentación oficial [16] encontramos información sobre diversos métodos, que podrían ayudar a añadir nuevas funcionalidades al módulo.

Por otro lado, en este subsistema, también tenemos la funcionalidad de realizar una lectura masiva de eventos, permitiéndonos, obtener la funcionalidad de sacar estadísticas de nuestros sistemas, ya que estos eventos contendrán el estado en cada momento del sistema. Esta funcionalidad la usa el subsistema "eChartsService" como veremos a continuación. Se divide en dos pasos: primero, obtiene los offsets, tanto el inferior como el superior de la partición que se use; en segundo lugar, se crea un consumidor, mediante la clase KafkaConsumer(org.apache.kafka.clients.consumer) y se realiza la lectura de eventos. Mediante una variable del application.yml se puede configurar una cantidad máxima de eventos si se desea para evitar posibles problemas de memoria.

```
final Properties readerProps = new Properties();
readerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, kafkaServer);
readerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, ↩
    ByteArrayDeserializer.class);
readerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, ↩
    ByteArrayDeserializer.class);
(...)
KafkaConsumer<byte[], byte[]> consumer = new KafkaConsumer<>(↩
    readerProps)
```

En el fragmento de código superior, vemos la configuración requerida y la creación de nuestro consumidor, también podemos ver como la lectura de los eventos se realiza en bytes. Una vez que tenemos los bytes corresponidentes a cada evento estos son transformados al objeto Java corresponidente de nuestro Evento:

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Event {
  String eventId;
  String eventName;
  Timestamp timestampCreation;
  String author;
  int version;
  String eventDataFormat;
  String payLoad;

}
```

Aquí podemos ver la estructura implementada en Java. Sobre el nombre de la clase podemos ver un conjunto de etiquetas correspondiente a la dependecia de lombook. Esta dependecia es de gran ayuda a la hora de desarrollar. Se encarga de generar de manera automática los getters y setters, mediante @Data y los constructores de la clase mediante @AllArgsConstructor y @NoArgsConstructor. Otro uso que se le da a esta dependecia en el conjunto del módulo es para logs, y de esta forma evitar el uso de escritura en la consola, esto lo hacemos mediante

la etiqueta @Slf4j, que nos genera una variable log que nos permite generar estos logs de error, advertencia, información o debug.

Subsistema eChartsService

Por último tenemos el subsistema que se encargará de generar los dashboard con la información de nuestro sistema. Su función principal es coger los datos generados por el subsistema anterior y realizar un filtrado de toda la información obtenida y mostrar esta información de una manera visual y que sea entendible para todo tipos de usuarios.

Es importante mencionar que este susbistema para la mayoría de estadísticas será independiente para cada situación. Se han realizado 3 estadisticas de ejemplo que se pueden sacar gracias a la utilización de la estrucutra del evento anteriormente generada, y que serán válidas siempre que se use esta estructura. En estos casos no se hace uso de un analisis del payload que es el que contiene la información del estado del sistema.

Para mostrar esta información de manera visual se ha utilizado la tecnologia *Apache eCharts* debido a su fácil integración y que al poder generarlos mediante javaScript nos permite una integración automática con nuestro API Rest sin necesidad de tener un frontal web. A continuación dos dashboard de los que han sido implementados:

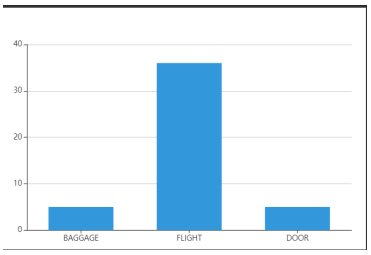

Figura 6.2: Gráfico de barras de los tipos de eventos

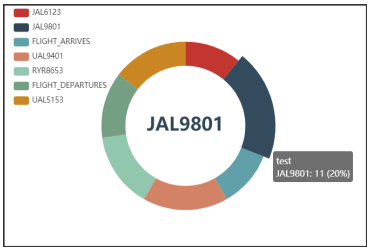

Figura 6.3: Gráfico circular de los tipos de eventos

En la figura 6.2 vemos los 3 tipos de eventos que nos genera nuestra arquitectura de ejemplo, correspondientes a vuelos, equipaje y puerta de embarque. Esto lo podemos obtener mediante una petición GET al endpoint correspondiente. En la figura 6.3 en lugar de usar el parámetro del tipo de eventos, usamos el parámetro de AUTHOR, que en nuestro ejemplo hace referencia al avión que genera cada evento.

El último dashboard que se ha implememtado, nos permite visualizar la cantidad de eventos que se generan en cada momento (por hora, minuto o segundo) según el formato que se desee. Para hacer los gráficos hemos ejecutado el simulador poco tiempo, se ha configurado para que nos muestre la cantidad de eventos por segundo, este es el resultado:

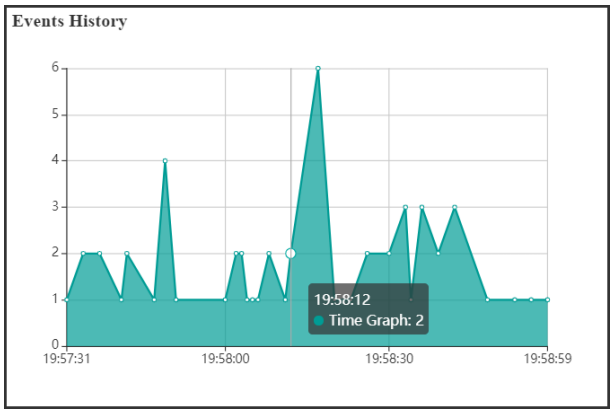

Figura 6.4: Gráfico circular de los tipos de eventos

Vemos que son dashboard bastante genéricos pero nos pueden ayudar a obtener gran información sobre nuestras arquitectura EDA y si todo está funcionando como debería. A partir de esto, usándolo como base, podemos sacar todo tipo de gráficos que deseemos mediante un buen filtrado y ordenación de la información de los eventos.

Podemos ver un ejemplo del código para generar estos dashboard en el anexo B.2 junto con el método que genera la API por la que se devuelven los dashboard, con la ayuda de la dependecia de Thymeleaf.

En el anexo D.2 podemos ver el listado de tecnologías que utiliza este módulo de estadísticas.

7

Conclusiones y trabajo futuro

7.1 Conclusiones

Con este trabajo de Fin de Grado se ha conseguido solucionar una necesidad real propuesta por el departamento de soluciones propias de Minsait. Necesitaban completar su documentación interna con un análisis de EDA. Al finalizar mis prácticas me lo propusieron, y viendo que podía ser una buena oportunidad para aumentar mis conocimientos de nuevas tecnologías desconocidas para mi decidí aceptarlo.

El trabajo ha consistido principalmente en una análisis de EDA, buscando la mayor cantidad de información posible de distintas fuentes, y a partir de esta información generar esta documentación. Con ella se pretendía generar algo así como una estandarización y un lugar donde poder consultar información por parte de los desarrolladores sobre dudas que puedan surgir durante su implantación. Para esto he de mencionar que en la web del departamento también se ha publicado una documentación similar, algo más simplificada, para que compañeros de la empresa puedan consultar.

A parte de la documentación también se ha desarrollado un ejemplo que muestra el funcionamiento de la arquitectura en una situación lo más real posible, es decir, con los distintos servicios desplegados en una máquina del departamento(Azure). A través de él se pretende mostrar que es posible realizar la comunicación entre servicios sin usar el protocolo REST, que es como actualmente se realizan la mayor parte de comunicaciones. Además, se ha implementado un módulo que permitirá explotar una de las utilidades que nos da EDA, la generación de estadísticas. Este módulo actuará como acelerador, ya que no es un módulo final, sino un punto de partida. Ambos, tanto el ejemplo como el módulo fueron presentados en el Spring Review del departamento junto con otros proyectos que se han ido realizando desde el departamento por otros compañeros.

También me gustaría mencionar que el desarrollo de esta memoria se ha realizado mediante la herramienta LaTeX. Hasta este momento no había realizado nada con esta herramienta y vi en el TFG la oportunidad de aprender otra tecnología nueva y de esta manera comprobar la

potencia que nos ofrece para generar documentación de una manera simple y ordenada. Para ello he utilizado la plantilla ofrecida por Rafael Leira en `https://github.com/ralequi`.

7.2 Trabajo futuro

Por último, se presenta una serie de posibles trabajos futuros a realizar:

- **Comparativa tecnologías gestor de eventos** Debido a políticas de la empresa en este caso se ha optado por el uso de kafka, sin embargo existen una gran cantidad de tecnologías que pueden ser usadas como gestor de eventos, por lo que habría que realizar un análisis de cada una de ellas y de esta manera poder saber cual es el mejor gestor en función de las necesidades de cada proyecto.
- **Ampliación ejemplos** Consistiría en la realización de nuevos ejemplos, intentando de esta manera cubrir el resto de casos de uso mostrados en el documento. Así se ayudará a desarrolladores futuros a la hora de implementar EDA en sus sistemas.

Bibliografía

[1] "¿que son los microservicios?" [Online]. Available: https://www.redhat.com/es/topics/microservices/what-are-microservices

[2] R. O. Roland Barcia, Kyle Brown, "Guía para punto de vista de microservicios." [Online]. Available: https://www.ibm.com/downloads/cas/ODGVKQE7

[3] "¿qué son los microservicios?" [Online]. Available: https://aws.amazon.com/es/microservices/

[4] V. Manuel, "Comunicando microservicios con apache kafka." 2019. [Online]. Available: https://www.paradigmadigital.com/dev/comunicacion-microservicios-apache-kafka/

[5] "Applying event-driven architecture to modern application delivery use cases." [Online]. Available: https://www.gartner.com/en/documents/3913523

[6] "Evento." [Online]. Available: https://dle.rae.es/?w=evento

[7] V. Madrid, "La aventura de las arquitecturas event-driven," 2019. [Online]. Available: https://enmilocalfunciona.io/aprendiendo-serverless-framework-parte-4-arquitecturas-eda/

[8] ——, "La aventura de las arquitecturas event-driven," 2020. [Online]. Available: https://enmilocalfunciona.io/la-aventura-de-las-arquitecturas-event-driven-parte-3/

[9] "What is an event-driven architecture?" [Online]. Available: https://aws.amazon.com/es/event-driven-architecture/

[10] "draw.io." [Online]. Available: https://www.draw.io/

[11] J. Skowronski, "Best practices for event-driven microservice architecture," 2019. [Online]. Available: https://hackernoon.com/best-practices-for-event-driven-microservice-architecture-e034p21lk

[12] E. Yourdon and L. L. Constantine., *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design.* New York, N.Y.: Yourdon press, 1979.

[13] "Api." [Online]. Available: https://datos.gob.es/es/apidata

[14] "Streamsets data collector." [Online]. Available: https://streamsets.com/products/dataops-platform/data-collector/

[15] "Status code definitions." [Online]. Available: https://tools.ietf.org/html/rfc2616#section-10

[16] "Class adminclient." [Online]. Available: https://kafka.apache.org/23/javadoc/index.html?org/apache/kafka/clients/admin/AdminClient.html

[17] "Get started with echarts in 5 minutes." [Online]. Available: https://www.echartsjs.com/en/tutorial.html#Get%20Started%20with%20ECharts%20in%205%20minutes

Acrónimos

AMQP Advanced Message Queuing Protocol. 4

API application programming interface. 5

BBDD Base de datos. 17

CI/CD Continuous Integration/ Continuous Deployment. 3, 58

CPU Central Processing Unit. 11

EDA Event Driven Architecture. 1

EPS Escuela Politecnica Superior. I

HTTP Hypertext Transfer Protocol. 4

HTTPS Hypertext Transfer Protocol Secure. 4

JSON JavaScript Object Notation. 7

OP Onesait Platform. 24, 33, 35

RAM Random Access Memory. 11

REST Transferencia de estado representacional. 5

TCP Transmission Control Protocol. 4

TFG Trabajo Fin de Grado. 1

UAM Universidad Autonoma de Madrid. I

XML eXtensible Markup Language. 7

Anexos

A

CI/CD

Todos los servicios implementados, tanto los de los ejemplos con el módulos de monitorización y estadísticas, deben estar listos para ser desplegados en los servidores de la compañía(Azure), para ello se deben implementar dos ficheros, el dockerfile que servirá para generar la imagen docker y a través de ella el contenedor con nuestro servicio; por otro lado el JenkinsFile, que se encargará se automatizar el proceso(CI) de esta generación y por último desplegar nuestro servicio en GIT. A continuación se muestran los ficheros:

A.1 dockerfile

```
FROM openjdk:8-jdk-alpine

# Metadata
LABEL module.maintainer="***@indra.es" \
    module.name="EdaExampleSimulator"

ADD *-exec.jar app.jar

# logs folder
RUN mkdir -p /var/log/admin-logs && \
  mkdir ./target

# create onesait user/group
RUN addgroup -S onesait -g 433 && adduser -u 431 -S -g onesait -h /usr/↩
    local -s /sbin/nologin onesait

RUN chown -R onesait:onesait /usr/local && \
  chown -R onesait:onesait /var/log/admin-logs && \
  chown -R onesait:onesait ./target && \
  chown onesait:onesait app.jar && \
  chmod -R 777 ./target && \
  chmod -R 777 /var/log && \
  chmod -R 777 /usr/local

VOLUME ["/tmp", "/var/log/admin-logs"]

USER onesait

EXPOSE 8080

ENV JAVA_OPTS="$JAVA_OPTS -Xms1G -Xmx3G" \
  CONTEXT_PATH="/api" \
  KAFKA_SERVER=<server>:<port> \

ENTRYPOINT sh -c "java $JAVA_OPTS -Dspring.application.json=↩
    $ONESAIT_PROPERTIES -Djava.security.egd=file:/dev/./urandom -↩
    Dspring.profiles.active=docker -jar /app.jar" && 1
```

Con el dockerfile que podemos ver encima seremos capaces de generar una imagen docker de nuestro módulo, en la sección ENV se deberán añadir todas las variables de entorno que se usen, principalmente en los ejemplos se establecen las de las variables que establezcamos en el application.yml, realmente podríamos no poner nada, ya que duránte el despliegue podremos ponerlas o sustituir estas.

A.2 Jenkisfile

```
def img
pipeline {
  agent {
    kubernetes {
      defaultContainer 'jnlp'
      yamlFile 'sources/podTemplate.yaml'
    }
  }
  post {
    always {
      logstashSend failBuild: false, maxLines: 10000
    }

    failure {
      updateGitlabCommitStatus name: 'build', state: 'failed'

    }
    success {
      updateGitlabCommitStatus name: 'build', state: 'success'

    }
  }
  options {
    timestamps()
    gitLabConnection('GitLab')
  }
  stages {



    stage('Build, test and deploy') {
      // Replace the active when{} with this one when migrating to TBD
      //when { environment name: 'gitlabActionType', value: 'MERGE' }
      when {
        expression { env.gitlabBranch =~ /(release.*|^master$|^↩
            develop$)/ }
      }
      stages {
        stage('Build and package') {
          steps {
            sh 'env'
            container('maven') {
              dir('sources') {
                sh 'mkdir -p $HOME/.m2'
                sh 'cp settings.xml $HOME/.m2'
                sh 'mvn package -DskipTests=true'
              }
            }
          }
        }
```

```
        stage('Test') {
          steps {
            container('maven') {
              dir('sources') {
                sh 'mvn test'
              }
            }
          }
        }
      }

      stage('build_img') {
        when {
          expression { env.gitlabBranch =~ /(release.*|^master$|^develop$)/ }
        }
        steps {
            container('docker'){
              dir("sources"){
                script {
                  def projectPom = readMavenPom file: "pom.xml"
                  img = docker.build("${projectPom.artifactId}:${projectPom.version}", "-f docker/DockerFile ./target")
                }
              }
            }
        }

      }
      stage('deliver_img') {
        when {
          expression { env.gitlabBranch =~ /(release.*|^master$|^develop$)/ }
        }
        steps {
          container('docker') {
            script {
              docker.withRegistry('*******.azurecr.io', 'registry-onesait') {
                img.push()
                switch (env.gitlabBranch) {
                  case "develop":
                    img.push("dev")
                    break
                  case "release":
                    img.push("beta")
                    break
                  case "master":
                    img.push("latest")
                    break
                }
              }
            }
          }
        }}}}}
```

El código anterior corresponde al fichero Jenkinsfile, este será usado por el servidor Jenkins para construir nuestro projecto(.jar), generar la imagen docker y por último depositarla en el regisrtry de la compañía, para ellos irá pasando por las distintas etapas que hemos puesto en nuestro fichero(stage).

Por último necesitamos tener el proyecto subido a un repositorio GIT y tener este configurado para que cada vez que se realice un merge a las ramas seleccionadas, por lo general develop y master, sea notificado a jenkins y este comience a realizar el proceso. También se deberá notificar al encargado de DevOps asignado a nuestro departamento para que la primera vez nos genere las pipes de Jenkis, que nos generará en el servidor de jenkins nuestro pipe y "job" a los que será donde se envía la notificación de los merge.

B

Código

B.1 Configuración seguridad

En el siguiente bloque de código vemos la configuración que hay que añadir al application.yml para la librería de seguridad:

```
openplatform:
  api:
  baseurl: https://********.onesait.com
  auth:
    login.path: /oauth-server/oauth/token
    token:
    verify.path: /oauth-server/openplatform-oauth/check_token
    grant_type: password
    scope: openid
    clientId: <REALM_ID>
    password: <REALM_SECRET>
```

Por razones de privacidad y seguridad se ha omitido los parámetros comprometidos como la url donde este desplegada nuestra plataforma, que puede ser un nombre de dominio si se tiene DNS, una IP publica o incluso la dirección del contenedor en caso de que esté desplegado en la misma máquina que la plataforma. Ademas deberemos indicar el id del realm que queremos usar, que nos permitirá agrupar usuarios y el secret de este realm, esta información la podremos obtener desde el controlPanel de la plataforma si entramos con un usuario de administración.

En este otro vemos la clase en la cual configuramos la seguridad, indicando a que partes de la aplicación se puede acceder y a cuales no.

```
@Configuration
@Profile("DEV")
public class SecurityConfigAuthorization extends ↩
    ResourceServerConfigurerAdapter {

  @Override
  public void configure(HttpSecurity http) throws Exception {
    http
    .authorizeRequests().antMatchers("/swagger-ui**","/swagger-↩
        resources/**","/webjars/**,*").permitAll()
          .antMatchers("/*").access("hasRole('ROLE_USER')");
  }

}
```

B.2 eCharts

El código que necesitamos generar nosotros desde nuestro endpoint es el correspondiente a los ajustes del dashboard que desde la documentación de echarts [17] lo llaman option, esta configuración se la pasaremos a la instancia de echarts para que la interprete, un ejemplo es el siguiente:

```
let times = [[${list}]]
option = {
  color : [ "#009C95", "#21ba45" ],
  title : {
    text : 'Fuel History',
    textStyle : {
      fontFamily : 'lato'
    }
  },
  tooltip : {
    trigger : 'axis'
  },
  calculable : true,
  xAxis : [ {
    type : 'time',
    boundaryGap : false,
    axisLabel : {
      formatter : (function(value) {
        return moment(value).format('HH:mm:ss');
      })
    }
  } ],
  yAxis : [ {
    type : 'value'
  } ],
  series : [ {
    name : 'Time Graph',
    type : 'line',
    smooth : false,
    itemStyle : {
      normal : {
        areaStyle : {
          type : 'default'
        }
      }
    },
    data : times
  } ]
};
```

En la primera línea vemos la variable times, que es donde se añadiran todos los valores que tendrán que seguir un formato similar a este: "["2020-04-13T19:57:31.000+02:00",1]", donde en el primer elemento hace referencia a la fecha y en segundo lugar la cantidad de repeteciones. También vemos como esta variable le introducimos el valor "[[$list]]", esto se debe a que los valores los añadiremos mediante la biblioteca thymeleaf, que nos permite rellenar una plantilla de documento html, u otros tipos de documento, con los valores que nostros queramos, al estar usando Spring es bastante sencillo de implementar, tan solo deberemos de añadir la dependencia en el POM.xml(spring-boot-starter-thymeleaf) e implementar un controller como el siguiente:

```
@GetMapping("/eventNameBarGraph")
public String eventNameBarGraph(Model model) {
  Map<String, Integer> surveyMap;
  try {
    surveyMap = admin.getEventNameChart();
    model.addAttribute("surveyMap", surveyMap);
    return "barGraph";
  } catch (KafkaAdminException e) {
    return null;
  }
}
```

En este caso el código pertenece al diagrama de la figura 6.2, podemos ver que los datos se obtienen de un método de nuestro servicio de administración de kafka(Subsistema KafkaService) y lo guardamos en un Mapa cuya clave sera el nombre del evento y el valor la cantidad de apariciones. Esto lo introducimos en el Modelo, que será de donde thymeleaf saque los datos. Por último en el código podemos ver un return "barGraph", esto también es parte de la dependencia de thymeleaf, lo que hace es de manera automática, buscar la plantilla con ese nombre en src/resources/templates y en ella aplicar el modelo.

C

Documentación mediante Swagger 2

Con el fin de tener documentadas las APIs y además ofrecer una interfaz para facilitar el trabajo durante el desarrollo, se ha incluido en el módulo Swagger2, el cual nos permite documentar nuestras APIs de una manera muy sencilla mediante el uso de etiquetas, y que este genere una interfaz con todos los endpoints disponibles, con la información y que además nos permita realizar las peticiones de una manera rápida y sencilla. A continuación se muestra un ejemplo de como se realizaría esta documentación a nivel de código mediante etiquetas.

```
    @GetMapping("/listTopics")
  @ApiOperation(value = "list all topics")
  @ApiResponses(value = {
      @ApiResponse(code = 200, message = "Request Successful ", response ↩
          = ResponseEntity.class),
      @ApiResponse(code = 400, message = "Bad request, review the request↩
           param"),
      @ApiResponse(code = 401, message = "User not authorized"),
      @ApiResponse(code = 500, message = "Unexpected exception (Internal ↩
          Server Error)")})
  public ResponseEntity<Object> getTopics() {
    (...)
    }
```

A través de todas las etiquetas que se encuentran en el código, se genera un fichero json o yml, que podemos encontrar en la ruta "/v2/api-docs" de nuestro servicio, es este fichero el que es interpretado por swagger-ui para poder generar una interfaz con la documentación como la que vemos en la captura C.1. Esta es expuesta en la ruta "swagger-ui.html". También podemos ver como nos aparece una cabecera opcional de autorización, esto se debe a que en la configuración de Swagger se ha establecido que aparezca este campo como opcional para ayudar en el desarrollo futuro, aunque como veremos a continuación, aunque aparezca como opcional en este caso será necesario para acceder.

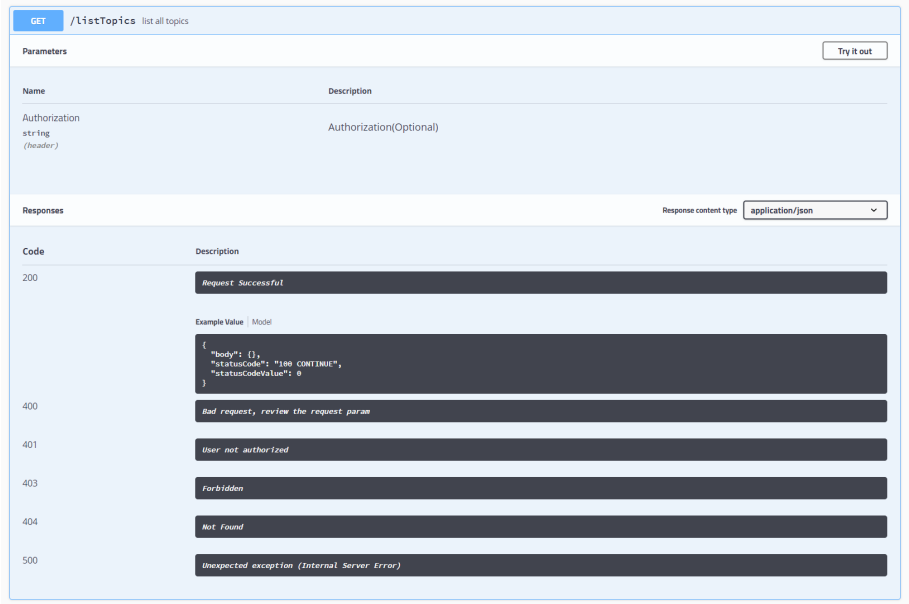

Figura C.1: Ejemplo de API en swagger

D

Tecnologías usadas módulos

A continuación podemos ver los listados de las tecnologías que usan tanto el ejemplo de EDA como el módulo de generación de estadísticas.

D.1 Listado tecnologías usadas en el ejemplo de EDA

- *Java 8* como lenguaje de programación principal.
- *Spring Boot* facilitar el desarrollo.
- *Kafka* como gestor de eventos, en este caso viene integrada dentro de la Onesait Platform.
- *Loombook* dependencia para facilitar el desarrollo en Java, automatización de getters, setters y constructores de clases Java.
- *Spring Tool Suite 4* IDE para el desarrollo, una extensión de Eclipse.
- *Node.js* Usado para generar el servidor del frontal.
- *HTML y JS* para la implementación de la parte web.
- *WebSocket* utilizado para abrir un canal de comunicación entre la parte back y front.
- *Swagger2* para la documentación de APIs y pruebas durante el desarrollo.
- *git* para el control de código.

D.2 Listado tecnologías usadas en el Módulo de estadísticas

- *Java 8* como lenguaje de programación principal.

- *Spring* especificamente Spring Boot para facilitar todo el desarrollo y Spring Secuity para dotar al módulo de seguridad.
- *eCharts* para la generación de dashboards en html.
- *Onesait Platform* plataforma de Minsait, exactamente se ha usado el identity manager y el oauth server.
- *Kafka* como gestor de eventos, en este caso viene integrada dentro de la Onesait Platform.
- *Loombook* dependencia para facilitar el desarrollo en Java, automatización de getters, setters y constructores de clases Java.
- *Spring Tool Suite 4* IDE para el desarrollo, una extensión de Eclipse.
- *Swagger2* para la documentación de APIs y pruebas durante el desarrollo.
- *Thymeleaf* motor de plantillas de XML/XHTML/HTML5.
- *Docker* para facilitar el despliege de aplicaciones, mediante la generaciones de imagenes y contenedores.
- *Jenkins* para el CI/CD que debe acompañar a la aplicación.
- *git* para el control de código.